Interactive Volume Rendering Based on Ray-Casting for Multi-core Architectures

Alexandre S. Nery[1], Nadia Nedjah[2], Felipe M.G. França[1], and Lech Jozwiak[3]

[1] LAM – Computer Architecture and Microelectronics Laboratory
Systems Engineering and Computer Science Program, COPPE
Universidade Federal do Rio de Janeiro, Brazil
[2] Department of Electronics Engineering and Telecommunications
Faculty of Engineering – Universidade do Estado do Rio de Janeiro, Brazil
[3] Department of Electrical Engineering – Electronic Systems
Eindhoven University of Technology, The Netherlands

Abstract. The Volume Ray-Casting rendering algorithm, often used to produce medical imaging, is a well-known algorithm and the underlying computation can be easily executed in parallel. This is due to the fact that the huge number of rays, used to sample the volumetric data, can be processed independently. However, the algorithm's performance may drop substantially when the complexity/size of the volumetric dataset increases. In this paper, we present three implementations of our parallel volume ray-casting algorithm in different multi-core architectures, such as CMPs, GPUs and MPSoCs. Furthermore, we show that using multi-GPUs, that perform in parallel, we can almost halve the rendering time. The performance and aspects of the three implementations are discussed.

1 Introduction

High performance visualization of 3-D datasets has always been one of the main goals in Computer Graphics. For 3-D volumetric datasets, such as those acquired by *Computer Tomography* (CT), the rendering process is generally known as Volume rendering. The volumetric dataset is usually composed of several stacked parallel slices (images) that form a 3-D volumetric dataset. There are different techniques to render 3-D volumetric datasets [9,2]. For instance, the *Marching Cubes* algorithm [11] is one approach to turn voxels samples into polygonal data, in order to create an actual set of 3-D primitives that can be rendered by regular GPUs pipeline. On the other hand, such technique may lead to a poor quality polygonal representation of the volume, because of the approximations that are performed to create the polygonal data. Thus, the Volume Ray-Casting algorithm is a better candidate for producing more accurate results [10,4]. Essentially, this algorithm samples equidistant points along the ray, inside the volumetric 3-D dataset. Each sample, i.e. Voxel (*volumetric pixel*), corresponds to a given color and opacity in one of the parallel slices of the dataset.

M. Daydé, O. Marques, and K. Nakajima (Eds.): VECPAR 2012, LNCS 7851, pp. 177–186, 2013.

The interpolated colors and opacities are merged through *compositing* to yield the color of the view-plane pixel through which a primary ray has been traversed. For instance, the algorithm can show specific parts of a human body volumetric dataset, such as bones or internal organs.

Interactive visualization of volumetric datasets is often difficult. The volume ray-casting performance can drop significantly as more complex datasets are used. On the other hand, volume ray-casting has a very high parallelization potential, as each ray can be processed independently, producing one corresponding pixel information. Therefore, there are consistent approaches to accelerate volume ray-casting with custom parallel architectures in hardware. In [6], a pipelined application-specific integrated circuit (ASIC) was created, fabricated in 0.35 μ technology and running at 125MHz. Such ASIC is capable of producing interactive frame-rates at some degree, since there are limitations regarding the size of the dataset (256^3 voxels). GPUs have recently become a good option for massively parallel processing of floating point data [8]. Thus, there are also approaches to accelerate volume ray casting using GPUs [5,3]. However, most of the volume ray-casting algorithms on GPU strongly depend on optimizations to achieve real-time rendering performance. For example, using *texture* or *constant* memories of the GPU to store frequently-used data can substantially improve the given algorithm performance, because of their much lower latency [8].

In this paper, we propose and discuss the implementations of our interactive, un-optimized and flexible parallel volume ray-casting algorithm with *supersampling* on three different multi-core architectures: Chip Multiprocessor (CMP), Graphics Processing Unit (GPU) and Multiprocessor System on Chip (MPSoC). The CMP implementation of the algorithm uses OpenMP, while the GPU implementation is CUDA-based. The MPSoC-based implementation on FPGA uses the shared DDR memory for synchronization. We extensively compare performance results of the GPU and OpenMP implementations, showing that the GPU implementation can reach interactive visualization, especially when a multi-GPU configuration is used. We also compared and analyzed the advantages of using multi-GPU configuration over a single-GPU configuration for varying workloads (number of primary rays). Finally, the MPSoC-based implementation on FPGA (Xilinx Virtex-5) shows the portability and scalability of the volume ray-casting algorithm, as several microprocessors (MicroBlaze [13] cores) can be mapped on the FPGA and run the algorithm in parallel. All the implementations have not been optimized to use any special features of the corresponding architectures.

The rest of this paper is organized as follows: Sections 2, 3 and 4 briefly explains the parallel Volume Ray-Casting algorithm in CMP, GPU and MPSoC. Then, Section 5 presents extensive performance results for the three implementations and compare them, while Section 6 draws the conclusion of this work.

2 Parallel Volume Ray-Casting in CMP

The OpenMP-based parallel volume ray-casting technique is presented in Algorithm 1, where we use a *for work-sharing construct*, that splits the execution of

the parallel section among the group of threads. Thus, iterations of the *for loop* are split across the group of threads. Therefore, in Algorithm 1, groups of rays are assigned to groups of threads for execution, leading to parallelization of rays. Each thread has its own private variables (i, j and s) that are used to control the loop iterations assigned to each thread in the beginning of the parallel section. Also, if *supersampling* is enabled, then each ray spawns a given number of neighbor sampling rays (i.e. in the vicinity of the primary ray), that are executed by the same thread. Thus, the color information of each pixel is measured from all the sampling rays, improving the overall quality of the resulting image.

3 Parallel Volume Ray-Casting in GPU

The CUDA-based parallel volume ray-casting is presented in Algorithm 2. In the CUDA programming model, a thread is actually a *lightweight thread*, because of their simplicity and faster context switching mechanism when compared to regular threads. Throughout this section, we refer to threads in CUDA as *lightweight threads*. In addition, the CUDA-based implementation in Algorithm 2 has not been optimized for GPU execution. For example, the kernel do not make use of *shared memory* or *texture memory*, that are usually employed to avoid global memory long latency penalties.

Algorithm 1. Volume Ray-Casting with OpenMP

Require: rays, uniform grid structure, 3-D dataset
Ensure: image

```
 1: # pragma omp parallel for private(i,j,s)
 2: for i = 0 to WIDTH do
 3:    for j = 0 to HEIGHT do
 4:       color pixel;
 5:       for s = 0 to N_SAMPLES do
 6:          ray ry ⇐ get_ray(i,j,s);
 7:          color aux ⇐ intersectGrid(grid, ry, dataset);
 8:          pixel ⇐ pixel + aux;
 9:       pixel ⇐ pixel / N_SAMPLES;
10:       image[i][j] ⇐ pixel ;
```

Modern general purpose GPUs are capable of executing many thousands of threads in parallel [8]. Thus, each thread can be assigned to a primary ray that crosses a pixel of the view-plane. The result is that a portion of the final image is going to be produced by a *block of threads* (one pixel per thread). The corresponding *CUDA Kernel* is presented in Algorithm 2, considering that all data transfers between the host and the GPU have been already performed. If *supersampling* is enabled, the thread will execute as many sampling rays as required, as shown in line 5 of Algorithm 2. The sampling rays are addressed in column chunks, as shown in line 6.

Algorithm 2. Volume Ray-Casting CUDA–kernel

Require: rays, uniform grid structure, 3-D dataset
Ensure: image
1: ray ry;
2: i ⇐ blockDim.x * blockIdx.x + threadIdx.x;
3: j ⇐ blockDim.y * blockIdx.y + threadIdx.y;
4: color pixel;
5: **for** *samples* = 0 to N_SAMPLES **do**
6: ry ⇐ rays[i][j+samples];
7: color aux ⇐ intersectGrid(uniform grid, ry, dataset);
8: pixel ⇐ pixel + aux;
9: pixel ⇐ pixel / N_SAMPLES;
10: image[i][j] ⇐ c; {corresponding pixel color}

3.1 CUDA-Based Implementation Using Multi-GPUs

In this implementation, the same kernel shown in Algorithm 2 is executed by each GPU. However, the input rays are split among the GPUs, increasing even more the parallel processing of rays. In order to use two GPUs, a separate thread must be created to access each GPU, because one thread cannot control both GPUs at the same time. For that reason, we use OpenMP to create two threads, each one controlling one GPU. The same idea can be extended for more than two GPUs, if available. In the end, the results from both GPUs are merged by the host process into one single image.

4 Parallel Volume Ray-Casting in MPSoC

The MPSoC architecture consists of up to four Xilinx MicroBlaze [13] microprocessors running in parallel at 125MHz. They are connected to a shared DDR memory via a Xilinx Multi-Port Memory Controller (MPMC) [12]. One of the microprocessors is connected to a few communication peripherals, to enable input/output data transmission between the MPSoC and a host machine, as well as to enable access to the FPGA's flash memory. Thus, all the microprocessors must wait until the whole 3-D volume data is available for computation.

The parallel volume ray-casting implementation is presented in Algorithm 3, where iterations of the *for loop* are split across the microprocessors, as shown in line 2. Therefore, in Algorithm 3, groups of rays are assigned to different microprocessors, since rays can be processed independently from the others. Each microprocessor knows which data to read and to write, according to its own identification number (CPU_ID= $0, 1, 2$ or 3) and also according to the total number of enabled microprocessors (N_CPU= $1, 2, 3$ or 4), as shown in line 2 of Algorithm 3. Finally, at each loop iteration, an image pixel is produced, as shown in line 9 of Algorithm 3.

Algorithm 3. Volume Ray-Casting with MicroBlaze

Require: rays, uniform grid structure, 3-D dataset
Ensure: image

```
1: for (i = 0; i < IMG_WIDTH; i++) do
2:    for j = CPU_ID; j < IMG_HEIGHT; j ⇐ j + N_CPU) do
3:       color pixel;
4:       for s = 0 to N_SAMPLES do
5:          ray ry ⇐ get_ray(i,j,s);
6:          color aux ⇐ intersectGrid(grid, ry, dataset);
7:          pixel ⇐ pixel + aux;
8:       pixel ⇐ pixel / N_SAMPLES;
9:       image[i][j] ⇐ pixel ;
```

5 Experimental Results

In this section we present the experimental results on different datasets for each multi-core architecture implementation. The CUDA-based implementation was compiled using the CUDA Toolkit 4.0, while the OpenMP-based implementation was compiled in GCC 4.4.4. Up to two NVIDIA GTX 470 GPU were used for execution of the algorithm in CUDA, while a Core i7 960 Intel Multiprocessor (at 3.2 GHz) was used for the algorithm execution in OpenMP. The MPSoC-based architecture was synthesized in Xilinx EDK 13.1 for a Virtex-5 XC5VLX50T FPGA and the parallel algorithm implementation was compiled using MicroBlaze gcc compiler 4.1.2. All the execution time results are measured in seconds and the volumetric dataset (Fig. 1) used in this work is available in [1].

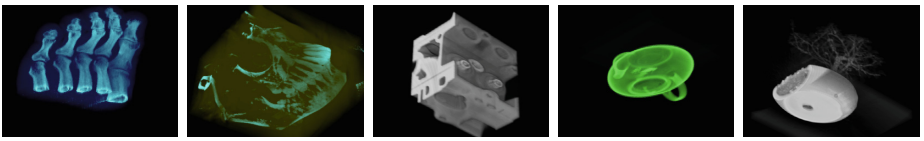


Fig. 1. Images produced by the proposed parallel volume ray-casting algorithm

Table 1. High-resolution execution times for eight different datasets

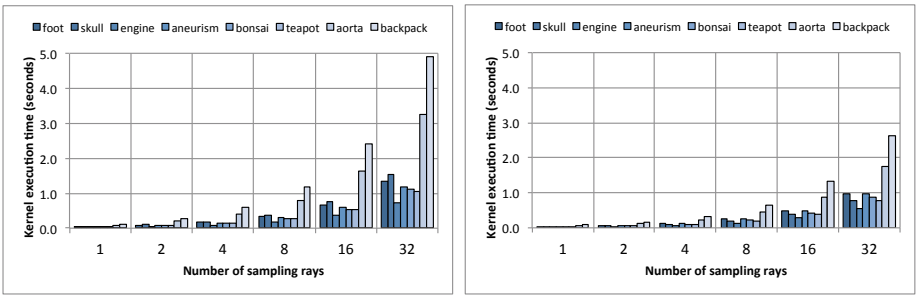
Data	Sampling rays, OpenMP Core i7						Sampling rays, single-GPU						Sampling rays, dual-GPU					
	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32
foot	2.54	4.68	9.02	18.30	34.58	69.75	0.03	0.08	0.17	0.34	0.67	1.35	0.03	0.06	0.12	0.24	0.48	0.96
skull	2.07	3.94	7.22	15.08	30.36	55.45	0.04	0.09	0.19	0.38	0.77	1.54	0.02	0.05	0.09	0.19	0.38	0.76
engine	2.14	4.12	8.63	16.81	33.42	66.45	0.02	0.04	0.09	0.18	0.37	0.74	0.02	0.03	0.07	0.14	0.28	0.56
aneurism	2.69	5.43	11.20	21.41	41.19	81.48	0.03	0.07	0.15	0.29	0.59	1.18	0.03	0.06	0.12	0.24	0.49	0.97
bonsai	2.23	4.19	7.41	14.65	30.16	55.78	0.03	0.06	0.14	0.27	0.53	1.11	0.03	0.05	0.11	0.22	0.44	0.87
teapot	2.58	4.57	8.72	16.84	38.36	73.00	0.03	0.06	0.13	0.26	0.53	1.06	0.02	0.05	0.09	0.19	0.38	0.77
aorta	6.19	12.22	24.08	47.94	95.76	192.46	0.08	0.19	0.39	0.80	1.62	3.26	0.06	0.12	0.23	0.45	0.88	1.76
backpack	6.36	12.51	24.97	49.45	99.22	198.32	0.12	0.29	0.58	1.20	2.43	4.91	0.08	0.17	0.33	0.66	1.32	2.62

5.1 High-Resolution Performance Results

For each volumetric dataset, the volume ray-casting algorithm was executed for 1280×800 primary rays, producing high-resolution images. In addition, the algorithm was executed with *supersampling* enabled, varying from 1 to 32 sampling rays per pixel. Therefore, up to 32 sampling rays were cast around the region of the primary ray pixel, producing smoother edges in the resulting image. The performance results are summarized in Table 1, for the OpenMP and the CUDA based implementations, using 1 and 2 GPUs, respectively. The MPSoC does not support high-resolution volume ray-casting processing because of memory limitations. Thus, its results are not included in Table 1.

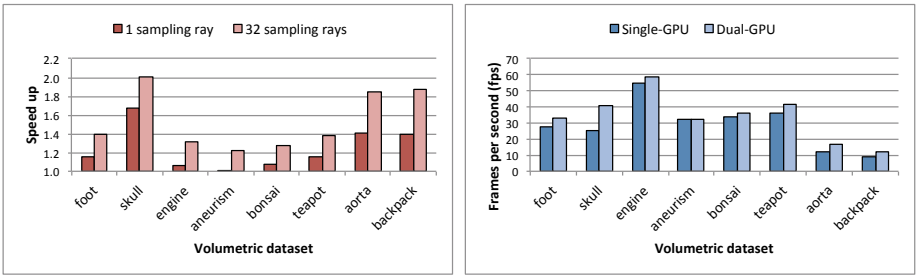
The OpenMP-based implementation uses 8 parallel threads, since the Core i7 microprocessor can execute up to eight parallel processes. The results in Table 1 show that even for one sampling ray, the performance is still not enough to ensure interactive visualization of the datasets. However, good results, i.e. image quality and interactive visualization, can still be obtained at lower resolutions, as fewer primary rays are processed. This will be shown in Section 5.2.

On the other hand, the GPU-based implementation results show that interactive visualization of volumetric datasets is possible even for high-resolution volume ray-casting. As depicted in Fig. 2a and 2b, the volume ray-casting execution time for every dataset is still below one second if up to 4 sampling rays are used, which corresponds to processing $1280 \times 800 \times 4$ rays, in total. Thus, more than one image, a.k.a frame, can be produced in one second, especially if less than four sampling rays are used. The dual-GPU implementation is around 90 times faster than the OpenMP implementation. Comparing the algorithm execution results using one and two GPUs, the performance is almost two times faster when two GPUs are employed instead of one. Also, observe that as more sampling rays are used, the performance gap increases, making the dual-GPU configuration a better candidate for high-quality interactive volume ray-casting, especially for complex datasets such as *aorta* and *backpack*, as shown in Fig. 3.



(a) Single-GPU, CUDA-based results. (b) Dual-GPU, CUDA-based results.

Fig. 2. GPU performance results in CUDA



(a) Single vs. Dual GPU speed up. (b) GPU frame rate (1 sampling ray).

Fig. 3. Acceleration rate using two GPUs and frames per second rate

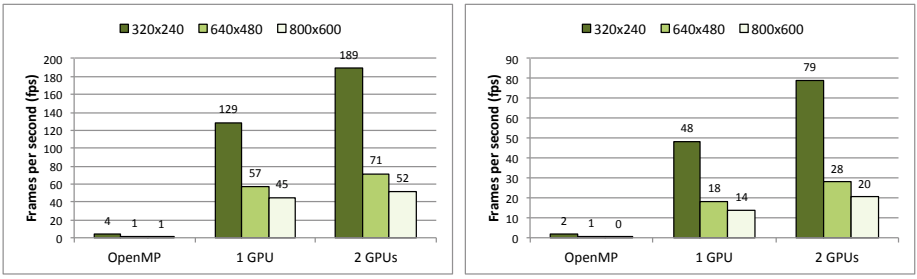
5.2 Lower-Resolution Performance Results

Lower resolution volume ray-casting can still provide a good trade-off between image quality and performance. In this section, we present some experimental results for the *foot* and *backpack* datasets, rendered in lower resolutions. The performance results are presented in Fig. 4, for one sampling ray.

It is clear that the GPU-based implementation can easily achieve real-time visualization (30 fps) of volumetric datasets when the resulting image resolution is decreased, which means that fewer primary rays are used to sample the volume data. For a simple dataset (*foot*), interactive visualization (around 60 fps) can be achieved even for higher resolutions, as in Fig.4a. On the other hand, the *backpack* dataset can achieve interactive visualization performance for very-low resolutions only, as depicted in Fig.4b. Moreover, the OpenMP-based implementation cannot provide real-time or interactive rendering yet. Thus, optimizations are necessary in order to improve the algorithm performance in OpenMP, as shown in [7].

5.3 MPSoC Synthesis and Performance Results

The MPSoC-based implementation results are shown in Fig. 5. Because of memory limitations of the FPGA we could render images of 640×480 pixels. Also,



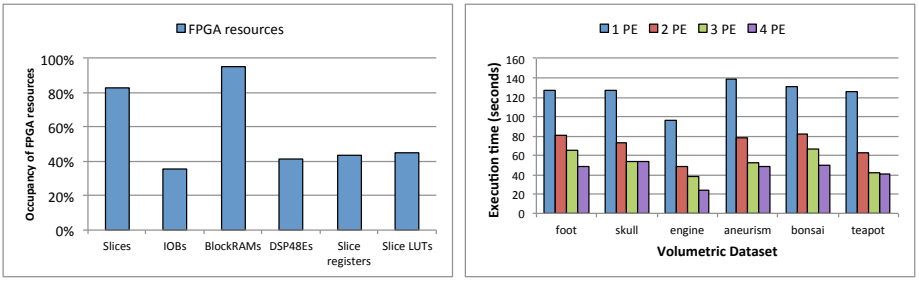
(a) *Foot* dataset low-resolution *fps*. (b) *Backpack* dataset low-resolution *fps*.

Fig. 4. Frames per second rendering rate for lower resolutions

we could not fit the *aorta* and *backpack* datasets in memory. In Fig. 5a, one can observe that almost all the FPGA slices are being used (82%), as well as the available BlockRAMs (95%). Because of that, we could only fit up to 4 microprocessors running in parallel. The high usage of BlockRAMs is due to the MPMC implementation of FIFOs for each input/output memory port, in order to improve timing and performance [12].

Performance and scalability results are shown in Fig. 5b. For most datasets the parallel algorithm execution time improves as more MicroBlaze microprocessors (Processing Elements - PEs) are being used in parallel. The MPMC FIFOs for the fourth microprocessor are implemented using shift register lookup tables instead of BlockRAMs, which can contribute to create stalls in the datapath and, hence, worsen the overall performance of the microprocessor.

Finally, it is clear that interactive performance is not yet achieved. However, an Application-Specific Integrated Circuit (ASIC) implementation of such application-specific MPSoC design, instead of FPGA, could most probably run faster, with lower area and power consumption, as in [6].



(a) FPGA area occupancy (4 PEs). (b) Execution times for 640×480 res.

Fig. 5. MPSoC synthesis and scalability, for up to 4 parallel microprocessors

6 Conclusions

In this paper, three un-optimized implementations of our volume ray-casting algorithm are discussed and compared. The GPU-based implementation is up to 90 times faster when a dual-GPU configuration is used, in comparison to the OpenMP-based implementation. One of the reasons for such speed up gain is because thousands of lightweight threads can be executed in parallel on GPU, while in the OpenMP-based implementation only 8 threads are executing in parallel. Furthermore, the overhead of changing between threads in GPU is much lower. The MPSoC-based implementation on a single Virtex-5 FPGA can execute up to four MicroBlaze microprocessors in parallel, running at 125MHz. As more microprocessors are used, the better is the performance achieved. However, interactive performance is not yet achieved, although in ASIC technology it could most probably run at higher frequencies, with more dedicated hardware.

Summing up, we demonstrated that the un-optimized GPU implementation of our volume ray-casting algorithm is able to deliver a high performance, between 60 and 90 times higher than that of our OpenMP-based implementation. For most datasets, high-resolution interactive visualization is achievable. Also, if we would make use of the texture and constant memories of the GPU, we would very likely achieve much higher frame rates, since the latency of these memories is much lower than the global memory latency. On the other hand, interactive performance may only be achieved in OpenMP unless several optimizations are applied to the algorithm. Furthermore, since the GPU implementation introduces more hardware overhead comparing to an MPSoC-based ASIC implementation, a MPSoC-based ASIC is expected to have lower area/power consumption.

References

1. Bartz: Volvis – volume library (2005), `http://www.volvis.org/` (last access May 2012)
2. Bhaniramka, P., Demange, Y.: Opengl volumizer: a toolkit for high quality volume rendering of large data sets. In: Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics, VVS 2002, pp. 45–54. IEEE Press, Piscataway (2002)
3. Cox, G., et al.: Exploring parallelism in volume ray casting: understanding the programming issues of multithreaded accelerators. In: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2012, pp. 64–73. ACM, New York (2012)
4. Wald, I., et al.: Faster isosurface ray tracing using implicit kd-trees. IEEE Transactions on Visualization and Computer Graphics 11(5), 562–572 (2005)
5. Mensmann, J., et al.: An advanced volume raycasting technique using gpu stream processing. In: GRAPP: International Conference on Computer Graphics Theory and Applications, pp. 190–198. INSTICC Press, Angers (2010)
6. Hanspeter, P., et al.: The volumepro real-time ray-casting system. In: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1999, pp. 251–260. ACM Press/Addison-Wesley Publishing Co., New York (1999)
7. Lee, V., et al.: Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA 2010, pp. 451–460. ACM, New York (2010)
8. Kirk, D., Hwu, W.M.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco (2010)
9. Lacroute, P., Levoy, M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, pp. 451–458. ACM, New York (1994)
10. Levoy, M.: Efficient ray tracing of volume data. ACM Trans. Graph. 9, 245–261 (1990)

11. Lorensen, W., Cline, H.: Marching cubes: A high resolution 3d surface construction algorithm. SIGGRAPH Comput. Graph. 21, 163–169 (1987)
12. Xilinx. Logicore ip multi-port memory controller (mpmc) v6.03.a, http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf (last access May 2012)
13. Xilinx. Microblaze reference, http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf (last access May 2012)