Hyper-Threaded Cache Coherent Raycasting

Sören Grimm Stefan Bruckner Armin Kanitsar Eduard Gröller

Institute of Computer Graphics and Algorithms
Vienna University of Technology *

ABSTRACT

Most volume rendering systems based on raycasting still suffer from inefficient CPU utilization and bad cache coherence. The recently introduced hyper-threading technology provides a solution to the first problem. This paper describes a raycasting system based on this new technology. To address the second problem the system is based on a bricked memory-layout. Bricking, however, requires an efficient addressing of data within and between blocks. We achieve this through two advanced address look-up tables.

Keywords: Hyper-threading, Simultaneous multi-threading, Raycasting, Cache Trashing, CPU Utilization, Bricking

1 INTRODUCTION

Three main volume rendering systems can be distinguished. The first ones are texture-mapping based; the second ones are special purpose hardware, e.g. VolumePro and Vizard, and finally there exist CPU based solutions ([5, 12, 14, 13, 10, 11]). Each of them has its advantages and drawbacks. Purely hardware based solutions provide real-time performance and high quality. But they are limited in there functionalities. In a commercial visualization system, the user has tools to process the data, e.g. filtering, segmentation, morphological operations, etc. If such operations are not supported by the hardware, they have do be performed on the CPU and data must be transferred back to the hardware. This transfer takes a lot of time. A system, where the user can modify the data interactively, and see the result immediately can only be achieved if the hardware supports it. These systems are mainly restricted by the offered functionality or one has to accept delays. This is the main reason why purely CPU based solutions are still commonly used. Another reason is the high flexibility. Many high-level optimization techniques have been developed to get high performance CPU solutions. Most of these techniques have one assumption in common: only parts of the data have to be visualized. This assumption is still valid, but the data delivered by new higher-resolution acquisition devices increases rapidly. This introduces two main issues. The first one is the enormous amount of data. The second one is, that there are more subtle details, which are difficult to handle in a high-level approach. One solution to deal with these issues is to focus on new technologies and on advanced low-level optimization techniques. This is, due to the fact, that most of the systems still do not utilize the available CPU resources efficiently. One of these new technologies is the recently introduced hyper-threading architecture by Intel. They introduced the first mainstream processor with simultaneous multi-threading support. Two threads simultaneously execute on one physical CPU, sharing all caches and execution units. In this paper we describe a raycasting system based on this new technology. To accomplish a high CPU utilization and cache coherence we based our system on a bricked volume layout. Bricking requires efficient addressing of the data. To achieve this, we present two advanced addressing techniques.

Section 2 surveys related work. Section 3 gives an overview of the used CPU hyper-threading architecture. Section 4 describes the whole system. It starts out with Section 4.1, which briefly describes a standard raycasting algorithm and its drawbacks. Section 4.2 presents our cache coherent algorithm and the two advanced addressing methods for a bricked volume layout. This algorithm is the basis for the hyper-threaded raycasting system presented in Section 4.3. In Section 4.4 the system is analyzed and the results are presented. Conclusions are given in Section 5.

2 RELATED WORK

The most prominent rendering approach which achieved high performance by using cache coherency is the Shear-Warp Factorization algorithm [8]. Cache coherency is achieved by doing re-sampling slice-wise and holding the data for each major viewing axis in memory. The main drawbacks are the low quality and the threefold memory usage. In contrast to this Knittel [7] achieved very high cache coherency by introducing a spread memory layout for fast access. He virtually locked all needed address look-up tables and color look-up tables into the cache. This leads to a rather high cache coherency, and therefore high CPU utilization. The memory usage, however, is increased by a factor of four. This memory storage requirement is way to high, considering that the maximum virtual address space of today's mainstream workstations is three Gigabyte. Therefore the maximum data-set size is limited by 3 Gigabyte / 4 = 768 MB. This seems to be an adequate size. But in most of the visualization systems, one can examine multiple data-sets. Furthermore, additional volumes or data-structures have to be kept in memory to support various operations like segmentation, filtering, etc. This makes it not practicable. Law and Yagel [9] proposed a parallel raycasting algorithm for a massively parallel processing system. They proved that appropriate data subdivision and distribution to the available caches, lead to high cache coherency. The scheme can be adapted to common single- and multi-processors. Using this scheme, leads to high cache coherency of all caches. But high CPU utilization is not inherent. Hyper-threading and advanced low-level optimizations turn out to be a solution to this utilization issue. We extended their basic scheme, so that the CPU utilization is significantly increased.

3 OVERVIEW OF THE HYPER-THREADING ARCHITECTURE

The explanation of hyper-threading will be focussed on the Pentium IV Xeon architecture, which we used for the realization of our raycasting system.

There are two main requirements to achieve high CPU utilization: First the execution units have to operate at full capacity; Sec-

*{grimm, bruckner, kanitsar, groeller}@cg.tuwien.ac.at. Favoritenstrasse 9-11, A-1040 Vienna, Austria

ond a high cache hit rate is desirable, which implies that no cache trashing occurs. The first condition will be fulfilled with hyper-threading. For the second condition we will define working sets so that they follow two known principals of locality, *temporal locality* - an item referenced now will be referenced again in the near future, and *spatial locality* - an item referenced now also causes his neighbors to be referenced. Before giving further details we take a closer look at the Pentium IV Xeon architecture.

3.1 Execution Resources

Figure 1 shows all the execution resources of the Pentium IV Xeon. They consist of one slow Arithmetic Logical Unit (ALU) for complex instructions, the Rapid Execution Unit which consists of two doubled-pumped ALUs for *simple* instructions and two double-pumped Address Generation Units (AGU), and the Floating Point Unit (FPU). The execution unit can execute at most three instructions at a time.

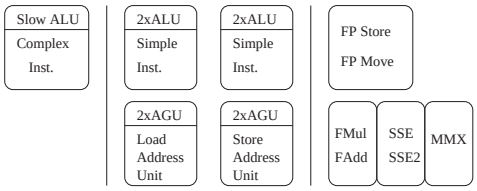

Figure 1: Execution Unit

3.2 Hyper-Threading

So far CPU designers tried to improve the CPU performance mainly by increasing the clock-rate. Currently Intel's record lies at 3 GHz, about 0.33 nanoseconds per clock-cycle. Achieving higher rates becomes more and more difficult due to physical laws and manufacture costs. Other directions to increase CPU performance were explored. The Pentium CPU was the first to allow the parallel execution of several instructions per clock-cycle. This feature alone was insufficient, because normally there are not enough sequential instructions which can be performed in parallel. To overcome this issue an out-of-order execution unit was introduced with the Pentium Pro. This unit reorders the instruction stream such that the CPU can execute more instructions in parallel. This concept is called instruction level parallelism (ILP). At first sight this is a very efficient solution, but studies have shown that in a typical application at most 2.5 instructions can be found to be executed in parallel.

Thus, there are still unused execution resources on the CPU. To use them, Intel recently introduced the hyper-threading technology for the Pentium IV Xeon. With this technology the CPU designers go one step further. Additionally to the instruction level parallelism, thread level parallelism (TLP) is introduced to identify even more instructions for parallel execution. Before, the out-of-order execution unit could choose from an instruction buffer of only one thread. Now, this buffer contains instructions of two threads which obviously increases the likelihood of finding data-independent instructions. This technology makes a single *physical* processor appear as two *logical* processors. It just duplicates the architectural state, while the physical execution resources and caches are shared (see Figure 2). In other words the CPU is capable to hold two thread contexts at the same time. The two threads are executed simultaneously on the same execution units, using the same caches. If one thread stalls due to a cache miss, the other one uses the idle execution resources.

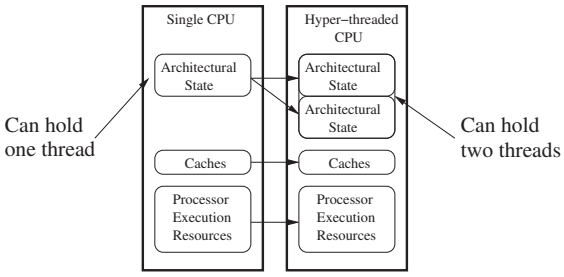

Figure 2: Hyper-threading technology duplicates the architectural state of the physical processor, providing two *logical* processors.

3.3 Cache

The cache hierarchy of a x86-based system is shown in Table 1. Going up the cache hierarchy towards the CPU, the caches get smaller and faster. In general if the CPU issues a load of a piece of data the request is propagated down the cache hierarchy until the requested data is found. It is very time consuming if the data is only found in a slow cache. This is due to the propagation itself as well as to the back propagation of data through all the caches.

Since the focus is on speed, frequent access to the slower caches has to be avoided. Accessing the slower caches, like hard disk and main memory, only once would be optimal. This is straightforwardly achieved for the hard disk level, as we assume that there is enough main memory to load all the data at once. To achieve this for the main memory is a lot trickier.

The Pentium IV Xeon has a 512KB level 2 cache using 64 Byte cache lines. It is 8-way associative and unified which means data and instructions share the same cache lines. The 8KB level 1 data cache is separated from the instructions and uses 64 Byte cache lines. Prefetching between the level 2 and level 1 cache is done in hardware. The read latency is seven clock cycles for the level 2 cache and two clock cycles for the level 1 data cache. The level 1 instruction cache is called Trace Execution cache. μOps are the smallest instruction units, into which x86-instructions are broken down. The instruction cache can hold 12,000 μOps and the size is about 92-96KB. Only the μOps of simple instructions are stored in this cache. For more complex instructions there is only a flag in the cache to signal that the μOps have to taken from the processor's micro code ROM.

In general when streaming linearly through data or instructions, caching does not make sense at all, because propagating the data through all levels of caches only increases the execution time. Caching does make sense if data and instructions are not only used once but used repeatedly with temporal and spatial locality. In other words working-sets are needed which fit into the caches and are used frequently. A more detailed description of the Pentium IV architecture can be found in [1, 2, 3, 4, 6].

Level	**Access**	**Typical Size**
register	1-3 ns	1 KB
level 1 cache	2-8 ns	8 KB - 128 KB
level 2 cache	5-12 ns	0.5 MB - 8 MB
main memory	10-60 ns	256 MB - 8 GB
hard disk	8 - 12 ms	100 GB - 200 GB

Table 1: Different levels and access times of the cache hierarchy on a x86-based system.

```
Number of rays is determined by the number of
pixels in the final image.
for    (all rays "r")
  while (ray "r" is still within the volume boundaries)
  {
      1.) Re-sample at position of ray r.
      2.) Gradient computation at position of ray r.
      3.) Shading at position of ray r.
      4.) Compositing at position of ray r.
      5.) Advance ray r.
  }
```

Figure 3: Standard raycasting algorithm.

4 HYPER-THREADED RAYCASTING SYSTEM

In this section we start out with a brief description of a straightforward raycasting algorithm and point out the main drawbacks. Then we present a cache coherent raycasting algorithm. Finally we describe a hyper-threaded raycasting system and give a performance analysis.

4.1 Standard Raycasting Algorithm

The standard volume raycasting algorithm on a linearly stored volume is given in Figure 3. For every pixel of the image plane a ray is shot through the volume and the volume data is re-sampled along this ray. At every re-sample position re-sampling, gradient computation, shading and compositing is done. From a performance point of view this work flow is very inefficient:

- The closer the neighboring rays are to each other, the higher the probability is that they partially process the same data. Given the fact that rays are shot one after the other, the same data *has to be read several times from main memory*, because the cache is not large enough to hold the processed data of a single ray.
- Different view directions cause a different amount of cache-line requests to load the necessary data from main memory which leads to a *varying frame-rate*.

These are the two main reasons, which lead to a bad CPU utilization. In the next sections these issues will be addressed and a solution is shown which performs very efficiently on an x86-based CPU.

4.2 Cache Coherent Raycasting Algorithm

In the following we describe a low-level optimized raycaster which utilizes the CPU very efficiently. We also explain the thereby developed new optimization techniques. With this raycasting system a detailed analysis of the hyper-threading technique is conducted to determine the achievable performance gain. To get optimal cache coherence, and high CPU utilization we based our system on Law's and Yagel's [9] raycasting method. They proposed a parallel raycasting algorithm for a massively parallel processing system (Cray T3D Supercomputer). The data was subdivided into small units and then evenly distributed among the processors, such that optimal cache coherence was achieved. This distribution scheme can still be used on current multi-processor systems. It can not be used for a hyper-threaded system. Therefore we are designing a significant different distribution scheme for the logical CPUs within one physical CPU.

```
Preprocessing:
1.) Create ordered list of blocks to process list.
2.) Add rays to blocks which are hit first.

Raycasting:
for (all blocks "b")
{
  while (block "b" contains rays to process)
  {
     for (all active rays "r" of block "b")
     {
       1.)  Re-sampling at position of ray r.
       2.)  Gradient computation at position of ray r.
       3.)  Shading at position of ray r.
       4.)  Compositing at position of ray r.
       5.)  Advance ray r.
       6.)  if(ray enters subsequent block)
            {
              (i) Remove ray from current block.
              (ii) Assign ray to the subsequent block.
            }
     }
  }
}
```

Figure 4: Block-wise raycasting algorithm.

In our system we also subdivide the volume data into small blocks. The blocks themselves are stored in xyz-order. The workflow of the algorithm, also shown in Figure 4, is as follows:
First a list of blocks is created such that it is sorted by the traversal order of the rays, and that each block has to be processed only once. Each block has initially an empty list of rays. At the beginning the rays are assigned to those blocks through which the rays enter the volume. Each of these blocks is then processed until all the rays enter subsequent blocks. If a ray enters a subsequent block, it is removed from the current block and assigned to the subsequent one. The subsequent blocks which contain the rays now are processed in the same manner. By this mechanism the rays are completely carried through the volume as soon as all the blocks are processed. These blocks basically define the data working-sets which were mentioned in section 3.3.

The evolution of CPU design shows that the CPU pipelines are getting longer and longer. This is very efficient as long as conditional branches do not initiate pipeline flushes. Once a long instruction pipeline is flushed there is a significant delay until it is refilled. Most of the present systems use branch prediction. The CPU normally assumes that if-branches will always be executed. It starts processing the if-branch before actually checking the outcome of the if-clause. If the if-clause returns false, the else-branch has to be executed. This means that the CPU flushes the pipeline and refills it with the else-branch. This is very time consuming. Using a block volume layout one will encounter this problem. The addressing of data in a block volume layout is more costly than in a linear volume layout. To address one data element, one has to address the block itself and the element within the block. In contrast to this addressing scheme, a linear volume can be seen as one large block. To address one sample it is enough to compute just one offset. In

algorithms like raycasting, which need to address a certain neighborhood of data in each processing step, the computation of two offsets instead of one has quite some performance impact. To avoid this performance penalty, one can construct an if-else statement. The if-clause consists of checking, if the needed data elements can be addressed within one block. If the outcome is true, the data elements can be addressed as fast as in a linear volume. If the outcome is false, the costly address calculations have to be done. On the one hand this simplifies address calculation, but on the other hand the involved if-else statement incurs pipeline flushes. In the following we take a look at this problem.

For raycasting, one can distinguish two major neighborhood access patterns. One is for re-sampling. The other one is for gradient computation. The latter will be solved generally for a 26-connected neighborhood access pattern. For the re-sampling computation the eight surrounding samples are needed. The necessary address computations in a linear volume layout are:

$$
\begin{aligned}
&\text{SampleOffset}_{i,j,k} && \rightarrow \text{i+j}\cdot D_x\text{+k}\cdot D_x\cdot D_y\\
&\text{SampleOffset}_{i+1,j,k} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1}\\
&\text{SampleOffset}_{i,j+1,k} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+}D_x\\
&\text{SampleOffset}_{i+1,j+1,k} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1+}D_x\\
&\text{SampleOffset}_{i,j,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+}D_x\cdot D_y\\
&\text{SampleOffset}_{i+1,j,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1+}D_x\cdot D_y\\
&\text{SampleOffset}_{i,j+1,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+}D_x\text{+}D_x\cdot D_y\\
&\text{SampleOffset}_{i+1,j+1,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1+}D_x\text{+}D_x\cdot D_y
\end{aligned}
$$

Thereby $D_{\{x,y,z\}}$ define the volume dimensions and i, j, k the integer parts of the current re-sample position in 3D. This addressing scheme is very efficient. Once the lower left sample is determined the other needed samples can be accessed just by adding an offset. In contrast to the linear volume addressing, the block volume addressing is:

if((i' $<$ BD_x-1) **and** (j' $<$ BD_y-1) **and** (k' $<$ BD_z-1))
{

$$
\begin{aligned}
&\text{SampleOffset}_{i,j,k} && \rightarrow \text{i'+j'}\cdot BD_x\text{+k'}\cdot BD_x\cdot BD_y\\
&\text{SampleOffset}_{i+1,j,k} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1}\\
&\text{SampleOffset}_{i,j+1,k} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+}BD_x\\
&\text{SampleOffset}_{i+1,j+1,k} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1+}BD_x\\
&\text{SampleOffset}_{i,j,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+}BD_x\cdot BD_y\\
&\text{SampleOffset}_{i+1,j,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1+}BD_x\cdot BD_y\\
&\text{SampleOffset}_{i,j+1,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+}BD_x\text{+}BD_x\cdot BD_y\\
&\text{SampleOffset}_{i+1,j+1,k+1} && \rightarrow \text{SampleOffset}_{i,j,k}\text{+1+}BD_x\text{+}BD_x\cdot BD_y
\end{aligned}
$$

}
else
{

$$
\begin{aligned}
&\text{SampleOffset}_{i,j,k} && \rightarrow \text{i'+j'}\cdot BD_x\text{+k'}\cdot BD_x\cdot BD_y\\
&\text{SampleOffset}_{i+1,j,k} && \rightarrow \textbf{ComputeOffset}\text{(i+1,j,k)}\\
&\text{SampleOffset}_{i,j+1,k} && \rightarrow \textbf{ComputeOffset}\text{(i,j+1,k)}\\
&\text{SampleOffset}_{i+1,j+1,k} && \rightarrow \textbf{ComputeOffset}\text{(i+1,j+1,k)}\\
&\text{SampleOffset}_{i,j,k+1} && \rightarrow \textbf{ComputeOffset}\text{(i,j,k+1)}\\
&\text{SampleOffset}_{i+1,j,k+1} && \rightarrow \textbf{ComputeOffset}\text{(i+1,j,k+1)}\\
&\text{SampleOffset}_{i,j+1,k+1} && \rightarrow \textbf{ComputeOffset}\text{(i,j+1,k+1)}\\
&\text{SampleOffset}_{i+1,j+1,k+1} && \rightarrow \textbf{ComputeOffset}\text{(i+1,j+1,k+1)}
\end{aligned}
$$

}

$$
\begin{aligned}
&\textbf{ComputeOffset}\text{(i,j,k)} && \rightarrow \text{BlkOffset}_{i,j,k}\cdot(BD_x\cdot BD_y\cdot BD_z) +\\
& && \quad \text{OffsetWithinBlk}_{i,j,k}\\
&\text{BlkOffset}_{i,j,k} && \rightarrow \text{(i''+j''}\cdot BVD_x\text{+k''}\cdot(BVD_x\cdot BVD_y))\\
&\text{OffsetWithinBlk}_{i,j,k} && \rightarrow \text{(i'+j'}\cdot BD_x\text{+k'}\cdot(BD_x\cdot BD_y)
\end{aligned}
$$

Thereby $BD_{\{x,y,z\}}$ define the block dimensions, $D_{\{x,y,z\}}$ define the volume dimensions. $BVD_{\{x,y,z\}}$ denote the block volume dimensions defined by $BVD_{\{x,y,z\}} = (D_{\{x,y,z\}}/BD_{\{x,y,z\}})$, i, j, and k are the integer parts of the current re-sample 3D-position, i', j', k' are defined by i' = (i mod BD_x), j' = (j mod BD_y), and k' = (k mod BD_z), and i'', j'', k'' are defined by i'' = (i div BD_x), j'' = (j div BD_y), and k'' = (k div BD_z).

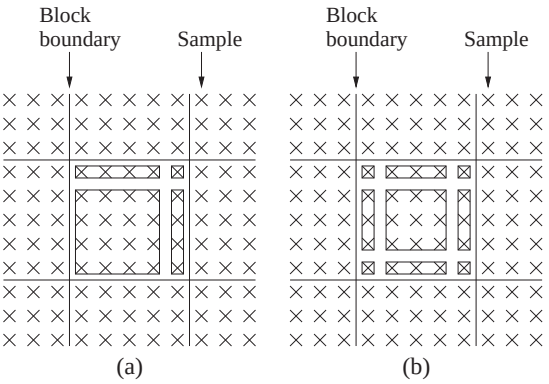

Figure 5: Sample position (i, j, k) is defined by the integer parts of the re-sample position. The sample positions (i, j, k) of a block are subdivided into subsets. The membership depends on the location of the adjacent samples. They are either in the same block or in one of the neighboring blocks. (a) Re-sampling: Four areas, because only samples to the right and to the top are accessed. (b) Gradient computation: Nine subsets, because samples in every direction are accessed.

To avoid the costly if-else statement and the expensive address computations, one can create a very small look-up table to address all the needed samples. The first sample location (i, j, k) is defined by the integer parts of the current re-sample position. The access pattern of adjacent samples during re-sampling is defined by accessing samples to the right, top, and back. The samples of a block can be subdivided into subsets. For the largest subset the seven adjacent samples of a sample (i, j, k) lie within the same block. The other subsets are defined by samples (i, j, k) on the border of the current block. Thereby the adjacent samples lie partially or completely within neighboring blocks. These other subsets are defined by the needed neighbor blocks to access all seven adjacent samples. The 2D case is illustrated in Figure 5(a). Thereby only samples to the right and to the top are needed, thus there are just four cases. Basically if the sample (i, j) lies on one or two of the block faces (top-, and right-face), neighboring blocks are needed. This can be mapped straightforwardly to the 3D case, by also taking into account the back-face. The thereby eight occurring cases are shown in Table 2.

Case	**i** $\in$	**j** $\in$	**k** $\in$
0	$\{0,...,BD_x-2\}$	$\{0,...,BD_y-2\}$	$\{0,...,BD_z-2\}$
1	$\{0,...,BD_x-2\}$	$\{0,...,BD_y-2\}$	BD_z-1
2	$\{0,...,BD_x-2\}$	BD_y-1	$\{0,...,BD_z-2\}$
3	$\{0,...,BD_x-2\}$	BD_y-1	BD_z-1
4	BD_x-1	$\{0,...,BD_y-2\}$	$\{0,...,BD_z-2\}$
5	BD_x-1	$\{0,...,BD_y-2\}$	BD_z-1
6	BD_x-1	BD_y-1	$\{0,...,BD_z-2\}$
7	BD_x-1	BD_y-1	BD_z-1

Table 2: The eight to distinguish neighbor block constellations.

As mentioned before the blocks are stored in xyz-order, therefore the necessary offset for the eight cases can be pre-computed and stored in a look-up table. The look-up table contains $8 \cdot 7 = 56$ offsets. We have eight cases, and for each sample (i, j, k) we need the offsets to its seven adjacent samples. The seven neighbors are accessed relative to the sample (i, j, k). Since each offset consists of four Bytes the table size is 224 Bytes. The tricky part is to address the look-up table efficiently. It can be achieved efficiently

Case	i & (BD$_x$ -1)	j & (BD$_y$ -1)	k & (BD$_z$ -1)	i + 1	j + 1	k + 1	i & ~(BD$_x$ -1)	j & ~(BD$_y$ -1)	k & ~(BD$_z$ -1)	(i+j +k) / Min(BD$_x$, BD$_y$, BD$_z$)
0	0-30	0-14	0-6	1-31	1-15	1-7	0	0	0	0
1	0-30	0-14	7	1-31	1-15	8	0	0	8	1
2	0-30	15	0-6	1-31	16	1-7	0	16	0	2
3	0-30	15	7	1-31	16	8	0	16	8	3
4	31	0-14	0-6	32	1-15	1-7	32	0	0	4
5	31	0-14	7	32	1-15	8	32	0	8	5
6	31	15	0-6	32	16	1-7	32	16	0	6
7	31	15	7	32	16	8	32	16	8	7

Table 3: Look-up table addressing for re-sampling. Thereby BD$_{\{x,y,z\}}$ = {32,16,8}.

if the block dimensions are a power of two, and a power of two apart. The second constraint can be removed by introducing a simple shift operation to virtually hold the constraint. To exemplify the algorithm it is assumed that the block dimensions are 32x16x8. The input of the look-up table addressing function is the sample position (i, j, k). As first step the block offset part from i, j, and k is extracted by anding the corresponding BD$_{\{x,y,z\}}$-1. The result can be seen in Table 3 second column. The next step is to add one to each of them. By this operation the current maximal possible value BD$_{\{x,y,z\}}$-1 is moved to BD$_{\{x,y,z\}}$. All the other possible values stay within the range [1,BD$_{\{x,y,z\}}$-1]. Then the resulting value is anded by the complement of BD$_{\{x,y,z\}}$-1. After this operation the input values are mapped to {0, BD$_{\{x,y,z\}}$}, as shown in Table 3, column four. The last and final step is to add the three values and divide the result by the minimum of the three block-dimensions BD$_{\{x,y,z\}}$, which maps the result into the range [0,7]. This last division can be exchanged by a shift operation. The final algorithm for a 32x16x8 block is:

SampleOffset$_{i,j,k}$ → **ComputeOffset**(i,j,k)
Index → ((((i&0x1F)+1)&0xE0)+
→ (((j&0x0F)+1)&0xF0)+
→ (((k&0x07)+1)&0xF8))>>3
SampleOffset$_{i+1,j,k}$ → SampleOffset$_{i,j,k}$+Lut[Index][0]
SampleOffset$_{i,j+1,k}$ → SampleOffset$_{i,j,k}$+Lut[Index][1]
SampleOffset$_{i+1,j+1,k}$ → SampleOffset$_{i,j,k}$+Lut[Index][2]
SampleOffset$_{i,j,k+1}$ → SampleOffset$_{i,j,k}$+Lut[Index][3]
SampleOffset$_{i+1,j,k+1}$ → SampleOffset$_{i,j,k}$+Lut[Index][4]
SampleOffset$_{i,j+1,k+1}$ → SampleOffset$_{i,j,k}$+Lut[Index][5]
SampleOffset$_{i+1,j+1,k+1}$ → SampleOffset$_{i,j,k}$+Lut[Index][6]

The *ComputeOffset* can be simplified, to just the offset calculation within one block. This is possible as the processing is done block-wise. Therefore the block-offset remains constant while processing one block.

Compared to the if-else solution which has the costly computation of two offsets in the else branch, we get a speed up of about 30%. The benefit varies, depending on the block dimensions. For a 32x32x32 block size the else-branch has to be executed in 10% of the cases and for a 16x16x16 block size in 18% of the cases. With larger block-sizes the percentage of the else-branch executions is smaller and therefore also the benefit decreases. But the focus is on small block-sizes anyway. For these sizes we reduced the overhead significantly. The other important benefit is, that it does not matter anymore where in the block adjacent samples are accessed. It is always done with constant computational time.

A similar approach can be done for the gradient computation. We present a general solution for a 26-connected neighborhood. Here we can, analogous to the re-sample case, distinguish 27 cases.

Case	i & (BD$_x$ -1)	j & (BD$_y$ -1)	k & (BD$_z$ -1)	i -1 & (BD$_x$ +BD$_x$ -1)	j -1 & (BD$_y$ +BD$_y$ -1)	k -1 & (BD$_z$ +BD$_z$ -1)	i | 1 + 1	j | 1 + 1	k | 1 + 1	i / BD$_x$	j / BD$_y$	k / BD$_z$	9·i +3·j+k
0	1-30	1-14	1-6	0-29	0-13	0-5	2-30	2-14	2-6	0	0	0	0
1	1-30	1-14	7	0-29	0-13	6	2-30	2-14	8	0	0	1	1
2	1-30	1-14	0	0-29	0-13	15	2-30	2-14	16	0	0	2	2
3	1-30	15	1-6	0-29	14	0-5	2-30	16	2-6	0	1	0	3
4	1-30	15	7	0-29	14	6	2-30	16	8	0	1	1	4
5	1-30	15	0	0-29	14	15	2-30	16	16	0	1	2	5
6	1-30	0	1-6	0-29	31	0-5	2-30	32	2-6	0	2	0	6
7	1-30	0	7	0-29	31	6	2-30	32	8	0	2	1	7
8	1-30	0	0	0-29	31	15	2-30	32	16	0	2	2	8
9	31	1-14	1-6	30	0-13	0-5	32	2-14	2-6	1	0	0	9
10	31	1-14	7	30	0-13	6	32	2-14	8	1	0	1	10
11	31	1-14	0	30	0-13	15	32	2-14	16	1	0	2	11
12	31	15	1-6	30	14	0-5	32	16	2-6	1	1	0	12
13	31	15	7	30	14	6	32	16	8	1	1	1	13
14	31	15	0	30	14	15	32	16	16	1	1	2	14
15	31	0	1-6	30	31	0-5	32	32	2-6	1	2	0	15
16	31	0	7	30	31	6	32	32	8	1	2	1	16
17	31	0	0	30	31	15	32	32	16	1	2	2	17
18	0	1-14	1-6	63	0-13	0-5	64	2-14	2-6	2	0	0	18
19	0	1-14	7	63	0-13	6	64	2-14	8	2	0	1	19
20	0	1-14	0	63	0-13	15	64	2-14	16	2	0	2	20
21	0	15	1-6	63	14	0-5	64	16	2-6	2	1	0	21
22	0	15	7	63	14	6	64	16	8	2	1	1	22
23	0	15	0	63	14	15	64	16	16	2	1	2	23
24	0	0	1-6	63	31	0-5	64	32	2-6	2	2	0	24
25	0	0	7	63	31	6	64	32	8	2	2	1	25
26	0	0	0	63	31	15	64	32	16	2	2	2	26

Table 4: Look-up table addressing for 26-connected neighborhood. Thereby BD$_{\{x,y,z\}}$ = {32,16,8}.

The 2D case is illustrated in Figure 5(b). Depending on the position of sample (i, j, k) a block is subdivided into 27 subsets. In contrast to the re-sample situation, additionally we have to handle sample positions on the bottom-, left-, and front faces.

The first step is to extract the block offset, by anding BD$_{\{x,y,z\}}$-1 as shown in Table 4, second column. Then we subtract one, and and with BD$_{\{x,y,z\}}$+BD$_{\{x,y,z\}}$-1, to separate the case if one or more components are zero. In other words zero is mapped to (2·BD$_{\{x,y,z\}}$-1) (Table 4, third column). All the other values stay within the range {0,...,BD$_{\{x,y,z\}}$-2}. The other case which has to be separated is the case if one or more of the components are BD$_{\{x,y,z\}}$-1. This can be done by adding one, after the previous minus one operation is undone by oring 1 without loosing the separation of the zero case. The result can be seen in Table 4, third column. Now all the cases are mapped to {0,1,2} to obtain a ternary-system. This is done by dividing the components by the corresponding block-dimensions. These divisions can be exchanged by fast shift operations. The last and final step is then 9·i+3·j+k to get unique values in the range of [0,26]. The final look-up table index computation for a 32x16x8 block is:

i' → ((i & 0x1F) - 1) & 0x3F
j' → ((j & 0x0F) - 1) & 0x1F
k' → ((k & 0x07) - 1) & 0x0F
i" → ((i' | 0x01) + 1) >> 5
j" → ((j' | 0x01) + 1) >> 4
k" → ((k' | 0x01) + 1) >> 3
Index → (i"·9+j"·3+k")

The benefit is a 40% speedup. The index computation is more costly compared to the re-sample lut. However, the percentage where the else-branch has to be executed nearly doubled. Therefore the more costly index computation is compensated by the higher percentage of costly cases. What we did not mentioned so far is the size of the look-up table. It is 27 cases · 26 offsets · 4Byte per offset = 2808 Bytes. This can be reduced by a factor of two due to symmetry reasons. Therefore we have a very small look-up table of about 1404 Bytes. Thus, the re-sample look-up table and the 26-connected neighborhood look-up table, fit into 2KB.

In the next section we will describe a hyper-threaded raycasting system based on this raycasting algorithm.

4.3 Hyper-Threaded Raycasting

In Section 4.2 we presented a block-based raycasting algorithm with a highly optimized addressing scheme. This algorithm enables now the analysis of hyper-threading. One of the main ideas to do raycasting block-wise is to have data working-sets which can be shared between two hyper-threads. This is very important since hyper-threads share caches.

The work-flow of the hyper-threaded raycasting system, illustrated in Figure 6, is as follows:
In the beginning seven treads, T1, ...,T7, are started. T1 is responsible for all the preprocessing. In particular it has to create lists of blocks which can be processed simultaneously. These lists are sorted by the traversal order of the rays. Each list is subdivided evenly by T1 and send to T2 and T3. After a list is send, T1 sleeps until its slaves are done. Then it sends the next list to process, and so on. T2 sends one block after the other to T4 and T5. T3 sends one block after the other to T6 and T7. After a block is send, they sleep until their slaves are done. Then they send the next block to process, and so on. T4, T5, T6, and T7 perform the actual raycasting. Thereby T4 and T5 simultaneously process one block, and T6 and T7 simultaneously process one block. By this mechanism all blocks are processed in the correct order. During this process the most critical part is when rays are assigned to subsequent blocks. As mentioned before each block holds a list of rays to process. It can happen that several threads want to assign rays to the same block. This problem is illustrated in Figure 7(a). The straight-forward solution would be to ensure that only one thread at a time can assign rays to a block. But this decreases the performance drastically. There are two common synchronization mechanism to choose from, one is a mutex the other a critical section. In general a critical section is used to synchronize threads, because it is a lot faster. Using hyper-threading however introduces the same synchronization issue as with multi-processing. Once two threads can be executed simultaneously a critical section is internally implemented as a slower mutex. This leads to an enormous performance decrease. This is due to the fact that the operating system has to check the mutex periodically to wake up waiting threads if the mutex is free. Performance can be increased by setting the spin loop count, in other words by changing the time the operating system is checking the mutex. Since this is still not optimal we developed a solution without any synchronization. The solution is illustrated in Figure 7(b). Each thread has its own ray-list in a block. Thus if a thread assigns a ray to a block, it is ensured that it is the only one which is writing into that list. But we have four threads and only two threads processing a single block. Therefore each thread has to read from two lists at a time. With this approach the rays of all lists are processed. Processing also includes removing rays from the lists. This is, however, not critical. It is ensured by the processing order of the blocks that a ray can never be assigned to a block which is currently processed. The last open question is the load-balancing. This is done by interleaving the rays during initialization.

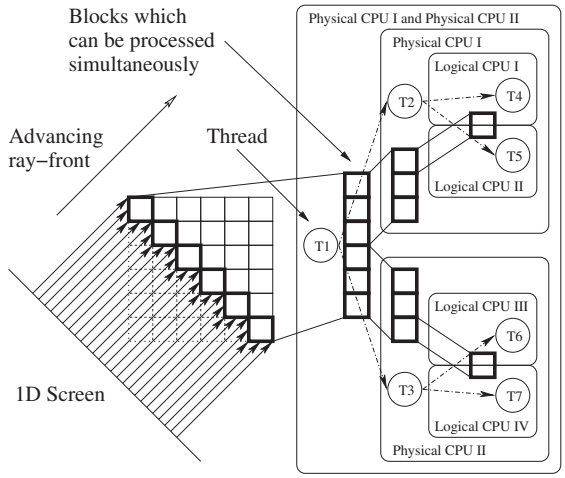

Figure 6: Hyper-threaded raycasting system

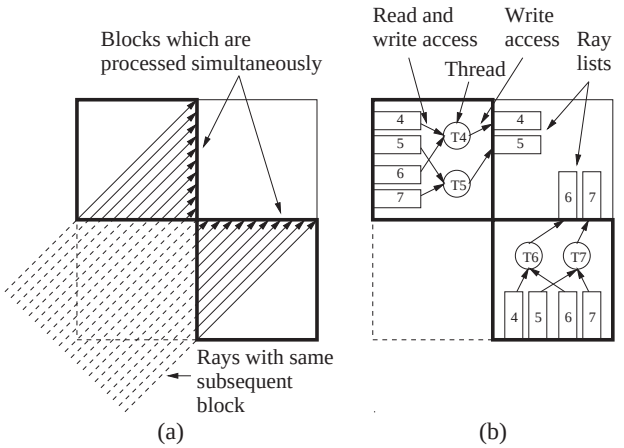

Figure 7: (a) Several threads assign rays to the same subsequent block. (b) Solution without synchronization.

4.4 Results

The achieved hyper-threading speedup is shown in Figure 8. The tested system is a Dual Pentium Xeon 2.4 GHz equipped with 1GB Rambus memory, and a GeForce IV graphics-card. Testing hyper-threading on only one CPU showed a speedup of 31%. Testing hyper-threading on two CPUs showed a similar speedup of 29%. The speedup is an average value. While changing the view-direction the cache hit-rate of the level 1 cache varies, and therefore also the speedup varies from 25% to 35%. Taking a block as a linear volume, then cache coherency between slices and within slices lead to this behavior.

Figure 9(a) shows the hyper-threading speedup according to different block sizes. The speedup significantly decreases with larger block sizes. Once the level 2 cache size is exceeded the two threads have to request data from main memory. Therefore the CPU execution units are less utilized. Very small block sizes suffer from a different problem. The data fits almost into the level 1 cache. Therefore one thread can utilize the execution units more efficiently, and the second thread is idle during this time. But the overall disadvantage is the inefficient usage of the level 2 cache.

Figure 9(b) shows the speedup achieved by block-wise raycasting. A worst-case comparison with respect to the view-direction, is shown. In case of small blocks the worst case is similar to the best case. Using large blocks shows enormous performance decreases depending on the view direction. The constant performance behavior of small blocks is one of the main advantages of a bricked volume layout. There is no view dependent performance variation anymore. The curves in both charts have an optimum at a block size of 64 KB. This number is also a good tradeoff between the needed cache space for ray-structures and sample data. The achieved 30% and 40% speedups of the two advanced addressing techniques for re-sampling and gradient computation are already included in the performance numbers. For a block size of 64 Kbyte

hyper-threading reduces computation time to approx. 70%. Block subdivision reduces computation time to approx. 35%. Combining the benefits of block subdivision and hyper-threading, we achieved an overall speedup factor of 4.0.

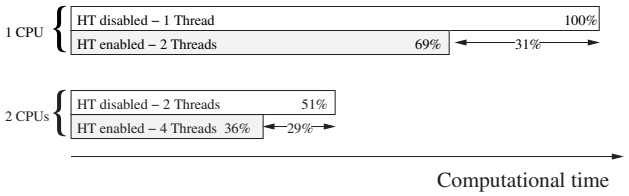

Figure 8: Hyper-threading speedup.

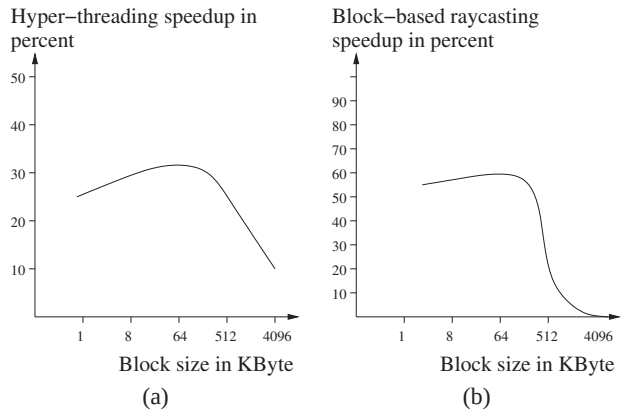

Figure 9: (a) Hyper-threading speedup for different block sizes. (b) Block-based raycasting speedup.

5 Conclusion

We have presented a raycasting system based on hyper-threading. For high cache coherency we used a bricked volume layout. Additionally, we developed two refined addressing schemes, such that data needed for re-sampling and gradient computations can be efficiently accessed.

For efficient use of hyper-threading we introduced a multi-threading scheme, such that two threads running on one physical CPU simultaneously process one data block. The results have proven that inefficient CPU utilization can be significantly reduced by using hyper-threading technology. The realization of the system showed that using this new technology is not straightforward. Systems have to be adapted to take advantage of this architecture. Most of today's used multi-threaded systems have to be redesigned. By just starting more threads one can encounter significant performance decrease instead of an increase. This is due to the fact, that hyper-treads share caches.

We achieved a significant speedup with our new addressing method in a bricked volume layout. The new addressing scheme can be used for any volume processing algorithm, which has to address adjacent samples. The results showed that conditional branches have quite some performance impact, due to the growing length of the CPU pipeline.

In conclusion, we have shown that advanced low-level optimizations lead to efficient CPU utilization and a significant speedup factor of 4.0.

6 Acknowledgements

The work presented in this publication has been funded by the ADAPT project (FFF-804544). ADAPT is supported by *Tiani Medgraph*, Vienna (http://www.tiani.com), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See http://www.cg.tuwien.ac.at/research/vis/adapt for further information on this project.

References

[1] Intel Cooperation. *IA-32 Intel Architecture Software Developer's Manual: Basic Architecture*. Intel, Order Number 245470-010, 2003.

[2] Intel Cooperation. *IA-32 Intel Architecture Software Developer's Manual: Basic Instruction Set Reference Manual*. Intel, Order Number 245472-010, 2003.

[3] Intel Cooperation. *Intel Cooperation, "IA-32 Intel Architecture Software Developer's Manual: System Programming Guide*. Intel, Order Number 245472-010, 2003.

[4] Intel Cooperation. *Intel Pentium 4 and Intel Xeon Prozessor Optimization, Reference Manual*. Intel, Order Number 248966-007, 2003.

[5] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and Arie Kaufmann. High quality volume rendering using texture mapping hardware. In *SIGGRAPH/Eurographics workshop on graphics hardware*, pages 69–76, 1998.

[6] Deborah T.Marr et al. *Hyper-Threading Technology Architecture and Microarchitecture*. Intel, www.intel.com, 2003.

[7] G. Knittel. The Ultravis system. In *SIGGRAPH Volume visualization and graphics sysmposium*, pages 71–78, 2000.

[8] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH*, pages 451–458, 1994.

[9] A. Law and R. Yagel. Multi-frame trashless ray casting with advancing ray-front. In *Proceedings of Graphics Interfaces*, pages 70–77, 1996.

[10] M. Meissner, S. Grimm, W. Strasser, J.Packer, and D. Latimer. Parallel volume rendering on a single-chip SIMD architecture. In *Symposium on parallel and large data isualization and graphics*, pages 107–113, 2001.

[11] B. Mora, J. Jessel, and R. Caubet. A new object order raycasting algorithm. In *Proceedings of Visualization*, pages 107–113, 2002.

[12] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. In *SIGGRAPH*, pages 251–260, 1999.

[13] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH*, pages 169–177, 1998.

[14] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Acceleration of ray-casting using 3d distance transforms. In *Proceeding of Visualization in Biomedical Computing*, pages 324–335, 1992.