UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

RNDR. ANTON VAŠKO

AUTOREFERÁT DIZERTAČNEJ PRÁCE

OPTIMIZATION OF VOLUME DATA FILTERING FOR SIMD-ENHANCED GENERAL-PURPOSE PROCESSORS

(OPTIMALIZÁCIA OBJEMOVÉHO FILTROVANIE PRE UNIVERZÁLNE PROCESORY OBSAHUJÚCE SIMD ROZŠÍRENIA)

na získanie akademického titulu philosophiae doctor

v odbore doktorandského štúdia
9.2.1. Informatika

BRATISLAVA 2011

Dizertačná práca bola vypracovaná v externej forme doktorandského štúdia na Katedre aplikovanej informatiky Fakulty matematiky, fyziky a informatiky Univerzity Komenského v Bratislave.

Predkladateľ: RNDr. Anton Vaško
FMFI UK
Mlynská dolina
842 48 Bratislava

Školiteľ: Prof. Ing. Miloš Šrámek, PhD.
FMFI UK
Mlynská dolina
842 48 Bratislava

Oponenti: .....
.....
.....
.....

.....
.....
.....
.....

.....
.....
.....
.....

Obhajoba dizertačnej práce sa koná ..... o ..... h pred komisiou pre obhajobu dizertačnej práce v odbore doktorandského štúdia vymenovanou predsedom odborovej komisie .....

9.2.1 Informatika

na Fakulte matematiky, fyziky a informatiky Univerzity Komenského
Mlynská dolina
842 48 Baratislava

Predseda odborovej komisie

.....

Abstrakt

Už od svojho zrodu sa počítače neustále vyvíjajú a zlepšujú. Ponúkajú stále viac a viac funkcií, fungujú rýchlejšie a vyžadujú menej miesta. Výkonnostné zlepšenia boli primárne dosahované zvyšovaním CPU frekvencie. V posledných rokoch však tieto narazili na technologické limity a tak ťažisko zvyšovania zmien prešlo na zmeny v architektúre a najmä na paralelizmus. Jasne to môžeme vidieť na poli grafických kariet (GPU), čo sú vlastne vysoko paralelizované špecializované obvody so svojou vlastnou RAM pamäťou. Aj v oblasti univerzálnych procesorov (GPP) môžeme pozorovať významné zmeny spočívajúce najmä v pridávaní nových a širších SIMD jednotiek a v používaní viacerých jadier.

V tejto práci prezentujeme efektívny algoritmus na konvolúciu (filtrovanie) na univerzálnych procesoroch. Filtrovanie sa často používa v spracovani objemových dát ako samostatný nástroj alebo ako významná časť v mnohých komplexných a časovo a výpočtovo náročných algoritmoch. Výkon filtrovania tak môže významne ovplyvňovať výkon jednotlivých algoritmov, v ktorých sa filtrovanie používa.

Objem spracovávaných dá často prevyšuje kapacitu pamäte bežných stolových počítačov. Preto pri filtrovaní (a vo všeobecnosti pri spracovaní dát) rozlišujeme dva prístupy. Prvý nazývame všetko-v-pamäti (AiM) a môže sa použiť ak sa filtrovaný objem vôjde do pamäte počítača. Druhý prístup je prúdové spracovanie (streaming) a zvyčajne sa používa ak veľkosť spracovávaného objemu prevýši veľkosť dostupnej pamäte. V tejto práci sa primárne zaoberáme výkonnostnou analýzou takzvaného separovateľného filtrovania (napr Gaussovské filtrovanie) v oboch prístupoch. Optimalizovali sme ich s ohľadom na efektívne využívanie SIMD jednotiek a vyrovnávacích pamätí rôznych úrovní. S týmito optimalizačnými technikami sme dosiahli urýchlenie o viac než jeden rád.

Abstract

Since their first construction, computers have been consistently improving. They offer more and more functionality, run faster and require smaller space. Performance improvements have been primarily achieved by increasing the CPU frequency. However, due to technological limits, engineers focus today on architectural changes of processors and especially on parallelism. This can be clearly seen in the area of graphics processing units (GPU), which are highly parallelized specialized circuits with their own dedicated RAM. In the area of general-purpose processors (GPP), we can also observe important changes, which reside particularly in adding new and wider SIMD units and in using multiple cores.

In this thesis we present efficient algorithms for convolution (filtering) on a GPP. Filtering is often used in volumetric data processing as a standalone tool or is an important part of numerous complex and time- and computationally-intensive algorithms. Performance of filtering thus may significantly influence performance of each algorithm it is used in.

Volume of the data to be processed often exceeds the available memory of the computer at hand. Therefore, we distinguish two approaches to volume filtering and data processing in general. The first one is called all-in-memory (AiM) and can be used if the volume to be filtered

fits into the memory of the computer. The second one is called streaming and is usually used if the size of the volume to be filtered exceeds the size of the available memory. In the thesis we primarily focused on analysis of the so called separable filtering (exemplified by the Gaussian and its derivatives) in both these approaches from the performance point of view. We optimized them with focus on efficient usage of the available SIMD units and efficient usage of the different cache levels of a processor. With these optimization techniques we achieved speedup of more than an order of magnitude.

Introduction

In 1965, Intel co-founder and visionary Gordon E. Moore predicted that the number of transistors on a chip will double about every two years and this trend would continue "for at least ten years" [Moo65]. Moore's law has driven the industry for over 50 years and is expected to be valid for at least another ten years. With the increasing number of transistors modern processor offer a lot of new functionality. For years, processor manufacturers consistently delivered increases in clock rates and instruction-level parallelism, so that single-threaded code executed faster on newer processors with no modifications. Nowadays the situation is a little bit different. Instead of hunting for higher frequencies, parallel computation has recently become more necessary and an easier way to follow in order to take full advantage of the gains allowed by Moore's law.

Current chips architectures contain two types of parallelism. The first one is called data parallelism and can be achieved by using the available SIMD (Single Instruction Multiple Data) instructions. As the name suggests, one SIMD instruction can process multiple data streams in parallel. The second form of parallelism is called task parallelism. It can be achieved by dividing a program into multiple tasks that can be run in parallel on available processor's cores, as current processors integrates two or more cores onto a single die or onto multiple dies in a single package. Another big and important change in computer architecture has begun in 2003 with introducing the 64 bit architecture called AMD64, x86_64 or simple x64.

All these architectural changes allow the old programs to be run with no or only small modifications. However, with this approach we cannot benefit from the gains the modern processors offer. The most of the old algorithms has therefore to be redesigned in order to efficiently utilize the available resources of modern CPUs.

Thesis Aims

The aim of this work is to analyze the performance of one basic operation which is very often used in volumetric data processing – the convolution/filtering. This operation is used for many purposes in multiple relatively simple algorithms (like edge detection or smoothing) and is a basic block of many complex algorithms (e.g. vessel enhancement). It is therefore very important to perform it as efficiently as possible. The naive implementations do not use the inherent parallelism available via different features of current desktop processors like SIMD or multiple-cores. Efficient implementation of convolution can cause that some complex algorithms, which are us-

ing it, will be executed more efficiently. As volumetric data processing is often used in medical environments, this can lead to popularization of some nowadays not used algorithms due to their high computational demands, which can in turn improve the computers' ability to better support doctors at their work.

The volume data to be filtered can be quite large today. Therefore, there is a need for a *streamed implementation* that is able to process a volumetric data that does not fit into main memory. As the streamed implementation usually introduces some overhead to the processing time, an *all-in-memory implementation (AiM)*, that can exploit the availability of the whole data in the main memory, is needed as well.

The commonly used filters are in floating-point precision (single or double). Due to the higher memory- and computational-requirements of a double-precision floating point version, the *single-precision* version should be preferred where possible.

Although being popular now, for multiple reasons we are not dealing with algorithm optimizations for GPU in this thesis. Some of the reasons are:

1. Limiting factor of GPU cards is that they are still equipped with insufficient amount of memory (512 MB on average).

2. High performance GPU cards are not yet standard on current desktop processors. However, high performance 64-bit CPUs are widely spread in nowadays desktop processors.

3. GPUs, due to massive parallelism available in them, are good at processing large amount data, where the work can be divided into many small parts. Filtering volume data matches this description only if the filter is long. As the filters are usually short, CPU seems to be better suited for this task.

Summarizing, the aims are:

- To analyze the basic operation in the volume data processing – filtering/convolution – for hotspots and bottlenecks.

- To design and implement algorithms for efficient filtering of volumetric data on current desktop processors. This implementation must involve efficient memory access and cache usage. Additionally, it should use the parallelism offered by the current desktop processors, namely data-level parallelism and multiprocessing.

- To implement separate versions for the 32-bit and 64-bit architectures. This is required due to the differences between these two architectures and because both of them are nowadays used.

Separable Filtering

Convolution is a mathematical operation on two functions (signals) f and g, producing third function (signal). It is defined as the integral of the product of the two functions one being

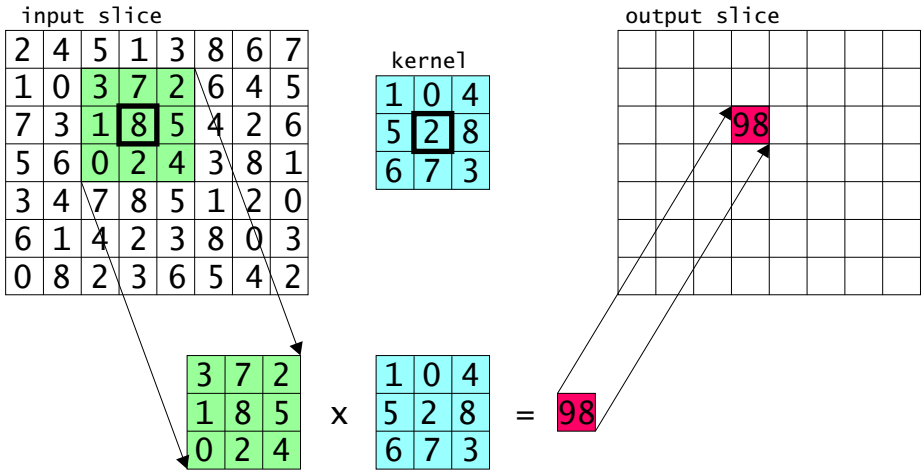

Figure 1: *Example of 2D filtering with a simple* 3×3 *2D filter.*

reversed and shifted:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) \cdot g(t - \tau) d\tau. \tag{1}$$

The discrete convolution of discrete functions is given by:

$$(f * g)[n] \stackrel{\text{def}}{=} \sum_{-\infty}^{\infty} f[m] \cdot g[n - m]. \tag{2}$$

In DSP (digital signal processing), convolution is often defined as transformation of the input signal x to the output signal y by means of the convolution kernel (filter) h:

$$y[n] = \sum_{j} h[j] \cdot x[n - j]. \tag{3}$$

Convolution is sometimes called filtering as well. For the purpose of this thesis, let us define the term **1D filtering** as follows:

$$y[n] = \sum_{j=-r}^{r} f[r + j] \cdot x[n + j], \tag{4}$$

where $r = \lfloor \frac{d}{2} \rfloor$ is radius of the filter, $f[0] \dots f[d - 1]$ are filter coefficients, x is input and y is output.

Similarly, **2D filtering** are **3D filtering** are defined as:

$$y[n_1, n_2] = \sum_{j=-r_2}^{r_2} \sum_{i=-r_1}^{r_1} f[r_1 + i, r_2 + j] \cdot x[n_1 + i, n_2 + j], \tag{5}$$

and

$$y[n_1, n_2, n_3] = \sum_{k=-r_3}^{r_3} \sum_{j=-r_2}^{r_2} \sum_{i=-r_1}^{r_1} f[r_1 + i, r_2 + j, r_3 + k] \cdot x[n_1 + i, n_2 + j, n_3 + k], \quad (6)$$

where $r_i = \lfloor \frac{d_i}{2} \rfloor$ are radii of the 2D/3D filter, f is filter, x is 2D/3D input and y is 2D/3D output. Figure 1 shows an example of 2D filtering for a simple 3×3 2D filter.

3D filtering is computationally a quite intensive task. Filtering of volumetric data with nz slices, where each slice has ny rows, each row having nx voxels, with a 3D filter of size $dx \times dy \times dz$, requires to perform $dx \cdot dy \cdot dz \cdot nx \cdot ny \cdot nz$ multiplications. Many multiplications can be saved if the 3D filter possesses some additional properties.

We say that 3D filter f is **separable** if there are filters f_1, f_2 and f_3 such that

$$\forall i \in \{0, \ldots, dx - 1\} \forall j \in \{0, \ldots, dy - 1\} \forall k \in \{0, \ldots, dz - 1\} : f[i, j, k] = f_1[i] \cdot f_2[j] \cdot f_3[k].$$

In this case, only $(dx + dy + dz) \cdot nx \cdot ny \cdot nz$ multiplications are performed when doing the convolution. Gaussian filter is an example of a 3D separable filter.

Memory Representation

The main disadvantage of the naive implementation resides in the cache-unfriendly memory access. Therefore, we propose a new technique called *Extended Volume(EV)* in AiM and *Extended Slab (ES)* in streaming to reduce the number of cache-misses.

When using EV, we allocate in memory "slightly more" space than the original volume requires. In order to filter the volume in direction D by 1D filter with size $2r + 1$ we need $2r$ neighboring values, r on the left (in the direction D) and r on the right. To avoid copying, it is sufficient to allocate an extended volume which has rz additional slices in the Z direction and each slice is extended by ry additional lines in the Y direction for storing the results of the intermediate filtering. This is just a negligible overhead, for a 512^3 volume and kernel of size 17^3 the size increase is only about 3.5%.

The original volume is then loaded into the top front area of this extended volume as shown in Figure 2, left. Then the filtering proceeds. The first pass, FilterXY, processes slices in the X and Y directions. It iterates through the slices of the volume backwards from rear to front. When processing a slice, it is first filtered in the X direction and the intermediate result is stored r slices and lines behind so that they do not overwrite the still required neighboring voxels. After this, the slice is filtered in the Y direction and stored in the top area of the same extended slice (see Figure 2, right). The second pass, FilterZ, iterates through the volume from front to back, filters voxels in the Z direction and stores the result in the original position in the top front area of the extended volume, where it is prepared for other operations.

SIMD

Next important step in algorithm optimization is vectorization. In filtering, we process volumes with voxels in single-precision floating point. Therefore, the we use the SSE instruction set for vectorization.

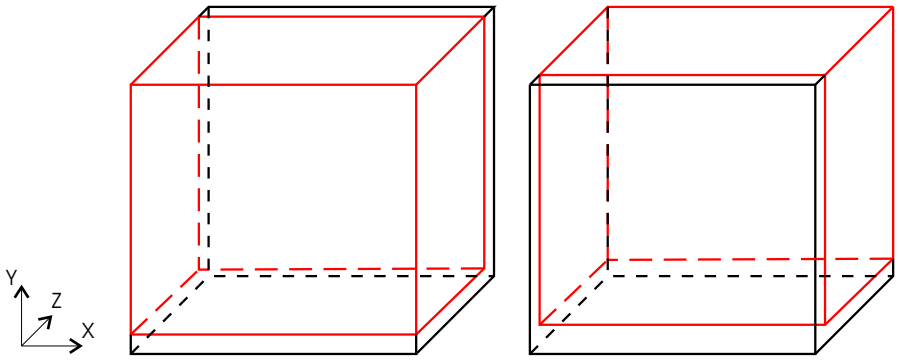

Figure 2: Position of the data volume in the extended volume during the filtering. Left—at the beginning before filtering. Right—after the FilterXY pass. After the FilterZ pass the filtered volume is positioned again in the top left area of the extended volume.

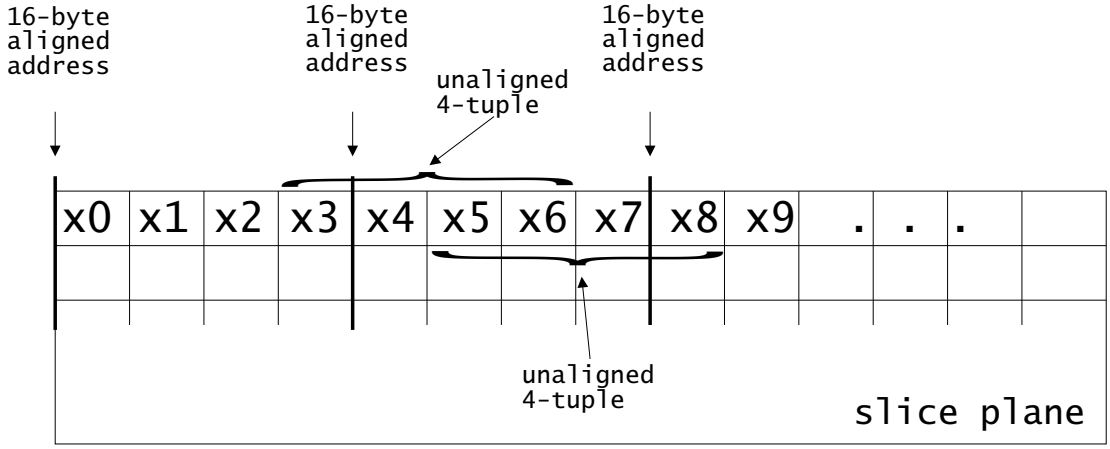

Figure 3: The (anti)symmetric filtering in the X direction. For filter of size 3, the four-tuples (x_3, x_4, x_5, x_6) and (x_5, x_6, x_7, x_8) lie in memory unaligned.

The main problem of vectorization in the X direction are unaligned reads. In order to avoid the performance penalties of unaligned reads we use two different techniques. In the case of symmetric/antisymmetric filtering [BDG+97], where we use the *vertical computational model*, we have to reorganize the input data using *data shuffling / swizzling*. Figure 3 demonstrates this for filter of size 3.

In the case of separable filtering, we use the *horizontal computational model* and the idea from [Int99].

Vectorization in the Y and Z direction is straightforward (see Figure 4 for filter of size 3 and Y direction). This technique (we call it *DirectVectorization*) also uses the vertical computational model. Although being straightforward, DirectVectorization suffer under many cache-misses due to cache-unfriendly memory accesses, especially when used in the Z direction, where the slice between two consecutive voxels is one whole slice. Therefore, we introduced a new technique,

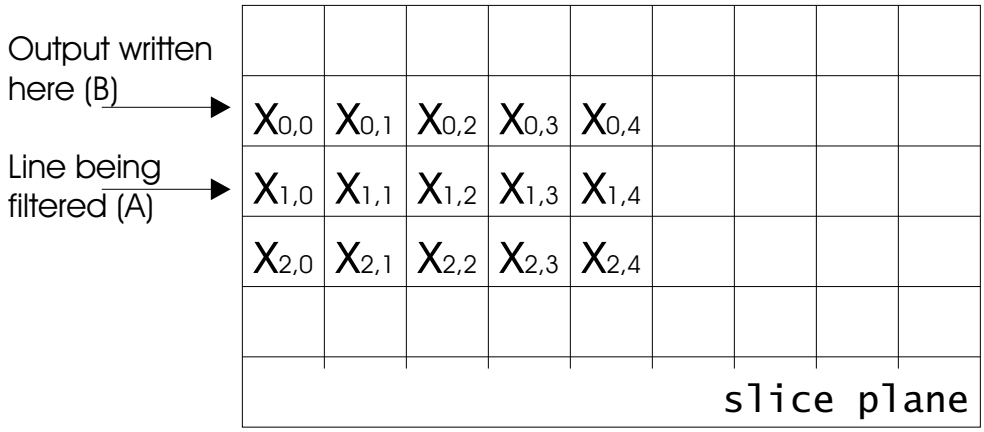

Figure 4: Vectorization in the Y direction for filter of size $d = 3$. When filtering a four-tuple A in a non-boundary line, the four-tuple B, which lies r lines above the line currently being filtered (in our case $r = 1$, i.e. the previous line) and is also used for filtering of A, is immediately rewritten with output values. This in-place filtering is only possible thanks to the Extended Volume/Extended Slab technique.

which is called the *BlockSize (BS)* technique.

The BlockSize technique is an alternative to the straightforward DirectVectorization technique.Although primarily designed for being used in the Z direction, the BlockSize (BS) technique can be used in the Y direction as well. Its goal is to combine both the cache-friendly memory access pattern (loop pattern) and efficient reuse of voxels already being in the L1 cache.

In the naive solution (in both Y and Z directions) voxels are copied from the volume to a temporary buffer. This is not needed if we use the EV or ES layout. The disadvantage of the copying clearly lies in the memory access pattern because the strides in the Y and Z directions are usually too high to be detected by the hardware prefetchers. However, the advantage of the copying (naive) approach is that the next iteration (in the Y or Z direction) can reuse the neighbouring voxels very well. On the other side, the advantage of DirectVectorization approach is that it uses cache-friendly memory access pattern, but if the volume is wide and/or the filter is long, the data brought into L1 cache will not be reused and must be loaded multiple times. Therefore, we combined the reusing of the neighbouring voxels from the naive version with the more cache-friendly memory access from the DirectVectorization approach as follows. In the Z direction, we divided the volume into multiple blocks ("subvolumes"). The depth of these blocks is always nz and the width of each block (perhaps except the last one) is *block size (BS)*. Every line from nz lines of a block consist of in memory neighbouring voxels. According to the block size BS and volume width, block may span one (Fig. 5a) or more (Fig. 5b) lines of a volume. The block size in this direction must not necessarily be smaller than the width of the volume. In such case, one block always spans more than one line in width. In the Y direction, each slice of the volume is divided into multiple blocks ("multicolumns"), see Fig. 5c. The block cannot be wider than the slice itself and the maximal possible block size in this direction is restricted to the volume width.

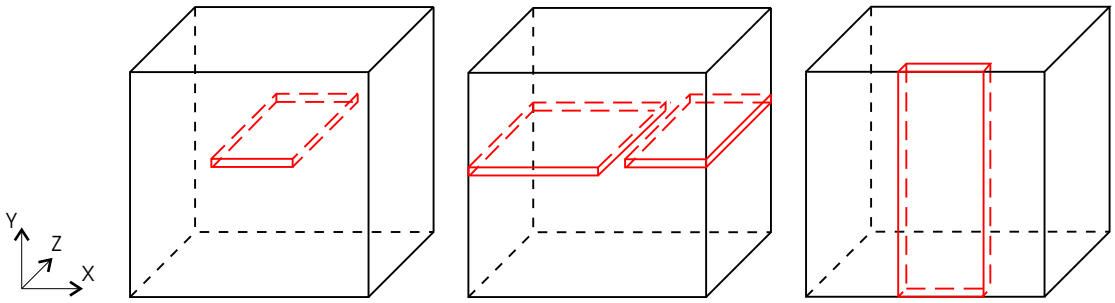

Figure 5: Volume division into blocks. In the Z direction, one block consists of $BS \times nz$ voxels and may span one (a) or more (b) lines of a volume in the Y directions, if BS is larger than the volume width or the block starts near the right boundary of a volume. In the Y direction (c) the block consists of $BS \times ny$ voxels and its width BS cannot exceed the volume width.

The volume is filtered block by block. As each block consists of multiple in memory neighbouring voxels, accessing block's voxels should be cache-friendly.

The pseudocode for the BlockSize algorithm may look as follows:

```
for(int xy=0;xy<nxy;xy+=BLOCK_SIZE)
{
    for(int z=0;z<nz;++z)
    {
        for(int x=0;x<BLOCK_SIZE;x+=4)
        {
            result={0.0, 0.0, 0.0, 0.0};
            for(int k=0;k<dz;++k)
            {
                val = get_voxels_by_slice_and_pos(z, xy+x);
                result += val * f[k];
            }
            store result;
        }//for x
    }//for z
}//for xy
```

If we supply the provided pseudocode with different arguments (e.g. `dy` instead of `dz`, `ny` instead of `nx` etc), then the same code can be used for both Y and Z directions as well.

Note that the block size in the Y direction may differ from the block size in the Z direction. Additionally, the width of the last block in each direction may be smaller than the block size. This happens if the volume width/slice size is not a multiple of the block size.

For good performance it is crucial to select the proper block size (BS) value. The best BS value is not a fixed value but depends on multiple factors (filter length, L1 cache size, voxel

type) and is calculated in run-time separately for the Y and Z directions as follows: we search for largest BS value, such that

$$BS \cdot dz \cdot \text{sizeof(float)} < L1\_CACHE\_SIZE \cdot LOAD\_FACTOR. \quad (7)$$

Additionally, BS must be multiple of 16 (i.e. 64B = cache line size) in order to minimize cache line splits. In our test applications, the $LOAD\_FACTOR$ is set to 0.9, i.e. 90%, as the L1 cache can hold also another data. In this way, we maximally reuse the data brought into the L1 cache.

Further Optimizations

All above-mentioned techniques are designed in such a way that they can be easily parallelized with the help of OpenMP [DM98]. Please note that using OpenMP does not automatically ensures the performance increase corresponding to number of available cores/processors.

When handling boundary voxels, we always use the mirroring approach. This approach requires that leading and trailing voxels of each row (X direction) or column (Y direction) or leading and trailing slices (Z direction) are to be mirrored. This can be easily implemented with an auxiliary buffer. We copy data in the corresponding direction in this buffer and mirror the leading and trailing voxels in the buffer. However, we would like to omit this copying into a buffer because it may be a bottleneck, especially when filtering in the cache-unfriendly Z direction.

Therefore, if we do not want to copy and mirror the data we have to copy and "mirror" the coefficients in a special way. The difference between data mirroring and coefficients mirroring is that mirroring of the coefficients can be done offline (prior to the whole calculation, i.e. only once) whereas mirroring of data must be always done online prior to filtering of a corresponding line or a column or a slice.

Mirroring of coefficients means that we prepare special sets of coefficients that will be used when handling boundary voxels. This preparation involves coefficients copying, shuffling and adding. Although mathematically correctly derived and implemented, using this technique we may obtain differences when comparing results with the non-optimized ones. The reason is that floating-point operations in our computers lack several math properties [Gol91]. For example, the distributive law between multiplication and addition does not necessarily hold. Therefore we state that both computed results, although being not exactly the same, can be thought as being correct.

Performance of filtering in the Y and Z directions can be extraordinary poor, especially when the volume sizes are power of two (POT) numbers. Unfortunately, such volumes are common and absolutely not rare, for example different scanner devices produce volume data sets with slice sizes being POTs.

POT problems occur when some of the consecutive lines (either in the Y or Z directions) compete in the L1/L2 cache for the same line. If this occurs, the cache memories are poorly reused and the performance significantly drops.

In order to solve this, we introduced additional padding (POTPADs) to our Extended Volume and Extended Slab layouts. The first padding is called *POTPADX* and is the line padding –

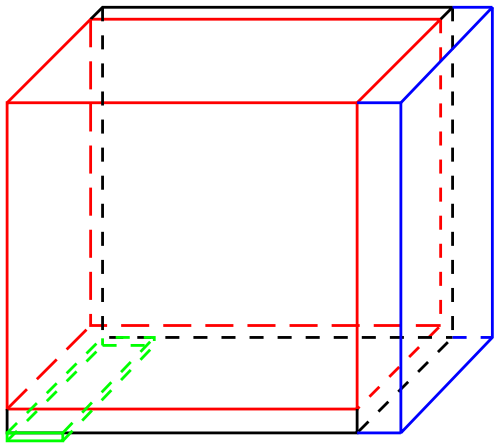

Figure 6: Extended volume with nonzero POTPADs. POTPADX voxels are added at the end of each line (the blue box on the right side). POTPADXY voxels are added at the end of each slice (the green box at the bottom).

when necessary we add POTPADX voxels at the end of each line of a slice. This will prevent the aliasing conflicts when filtering in the Y direction. However, even with padded lines the aliasing conflicts may happen when filtering in the Z direction. To prevent the conflicts in the Z direction as well, we introduced another padding – *POTPADXY*. POTPADXY actually means slice padding (i.e. it is padding between two consecutive slices) – we simply add POTPADXY voxels at the end of each (extended) slice. Both padding values depend on multiple factors and are determined in runtime. Figure 6 shows the Extended Volume with both POTPADs.

Results

We implemented all above-mentioned optimization techniques. For streaming, we used the f3dfilter library from the f3d [SD02] suite.

Our implementation was evaluated on a computer with the Intel Core2 Duo processor (E6550, 2.33GHz, 2 cores) with an L1 data cache of 32KB (per each core) that is 8-way set-associative and has a line size of 64 bytes. The 4096 KB L2 cache is 16-way set-associative with a 64-byte line size and is shared across both cores. The system was equipped with 2048MB of RAM.

Performance was measured using the cycle counters [Int10] as counting of cycles (also known as clockticks) provides a very precise tool for measuring the time that elapses between two different points in the execution of a program. In order to eliminate the effects of context switching, the K-best measurement scheme has been used as explained in [BO03].

As we are interested only in the optimization aspect of our implementations, we always measured only the key methods, that were subject of optimization. All disks operation, such as loading from/storing to disk were therefore omitted from measurements.

We filtered 16 volumes with voxels in the single-precision floating point representation (see Table 1). In order to achieve high performance, the core routines were implemented in assembly. Usually, they are specialized assembler versions for different filter sizes (3, 5, 7, ... 17) for

	Width [nx]	Height [ny]	Depth [nz]	Line size [in B]	Slice size [in KB]	Volume size [in MB]
vol0	64	64	64	256	16	1
vol1	80	80	80	320	25	2
vol2	128	128	128	512	64	8
vol3	144	144	144	576	81	11
vol4	256	256	256	1024	256	64
vol5	240	240	240	960	225	53
vol6	512	512	512	2048	1024	512
vol7	528	528	528	2112	1089	562
vol8	1024	1024	256	4096	4096	1024
vol9	1040	1040	240	4160	4225	990
vol10	2048	2048	64	8096	16384	1024
vol11	2032	2032	48	8128	16129	756
vol12	256	256	2048	1024	256	512
vol13	240	240	2032	960	225	446
vol14	1024	1024	1024	4096	4096	4096
vol15	2048	1216	1871	8096	9728	17775

Table 1: Dimensions of volumes used for testing filtering.

each vectorization technique we used. Additionally, we must distinguish between 32-bit and 64-bit versions assembler implementations, because the latter take benefits from more registers available in 64-bit processors.

Results obtained from 32-bit Windows are very similar to results from 32-bit Linux and the same holds for 64-bit Windows and Linux. Therefore, we present only the results from the Windows OS (both 32- and 64-bit).

Figure 7 shows the relative speedups of naive (no optimization) and SSE (SIMD with all described optimizations) versions obtained by running separable filtering of size 7^3 in the 32-bit environment. We can see that speedups are high in all directions, especially in the Y direction. SSE cannot cause such high speedups, the important role play here also additional (cache and memory related) optimization techniques.

Others filters achieve similar results. In the streamed environment, the results for the X and Y directions are very similar to those in AiM. However, in the Z direction, the speedups are significantly lower, especially when slab does not fits into the L2 cache.

Table 2 shows speedups obtained after enabling OpenMP. We can see that speedups vary from no speedup (e.g. 0.19) to high speedups (e.g. 2.07). Generally, with increasing filter and volume size the algorithm may benefit more from running filtering at multiple cores. In streaming, the X and Y directions behave similarly, however, in the Z direction OpenMP cannot help, because memory is here the main bottleneck.

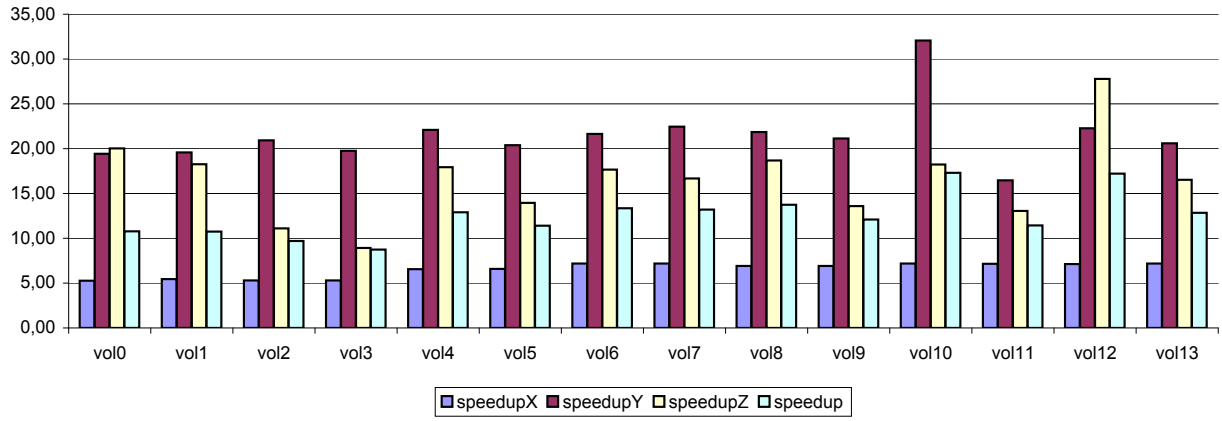

Figure 7: Measured speedups of naive (N) against SSE (SSE) of the separable filter of size 7^3 (all-in-memory version) on the 32-bit architecture.

Speedup	Separable Filter $7 \times 7 \times 7$				Separable Filter $31 \times 31 \times 31$			
[times]	FilterX	FilterY	FilterZ	Total	FilterX	FilterY	FilterZ	Total
vol0	0.13	0.64	2.09	0.19	0.47	1.15	1.16	0.43
vol1	0.63	0.73	2.13	0.76	0.30	1.14	2.15	0.54
vol2	0.58	0.92	1.76	0.73	0.93	1.71	1.66	1.09
vol3	0.59	1.25	1.62	0.86	1.17	1.64	2.04	1.47
vol4	1.21	1.50	0.95	1.13	1.54	1.90	1.74	1.68
vol5	1.47	1.59	0.98	1.29	1.63	1.95	1.96	1.78
vol6	1.09	1.85	0.98	1.14	1.83	1.90	1.73	1.81
vol7	1.12	1.74	0.99	1.15	1.85	1.85	2.63	2.07
vol8	1.01	1.19	0.97	1.03	1.90	1.92	1.83	1.88
vol9	1.01	1.13	0.97	1.02	1.91	2.11	1.75	1.91
vol10	1.06	0.98	0.97	1.01	1.92	2.02	1.74	1.89
vol11	1.07	0.96	0.98	1.00	1.94	2.23	2.05	2.06
vol12	1.34	1.67	1.01	1.24	1.75	1.90	1.73	1.78
vol13	1.31	1.56	0.99	1.21	1.81	1.99	1.86	1.87

Table 2: Speedups of the SSE versions of separable filter of sizes 7^3 and 31^3 respectively on the 32-bit architecture (all-in-memory) after enabling OpenMP (SSE without OpenMP against SSE with OpenMP).

Batch Streaming

If compare AiM filtering with the streamed one we find out that filtering in the Z directions performs poor in streaming. The reason is that streaming does not use the BlockSize technique in the Z direction because there is only minimal amount of slices available in slab. The main idea behind BatchStreaming is to use all efficient techniques from AiM also in streaming. This could be changed if there were more slices in the slab. Therefore, we modified processing of slices within the filter in such a way that we are processing slices in batches (hence the name Batch Streaming).

Let B ($B > 0$) denotes the batch size, i.e. the number of slices being processed at once in the Z direction. The slab thus consist of $dz + B - 1$ slices. The filter in BatchStreaming always turns between two phases. In the *upload phase* (the starting one) the filter is ready only for uploading. The incoming slices are processed (in the uploadSlice method) and stored in the slab. There are multiple uploads required until the slab is full or there are no more slices left for processing. Then the filter swaps its phase to the *download phase*. While swapping, *all B* slices in the slab may be processed either by one or as a one block. In this way, there may be no or only a little processing in the download phase, when the already processed slice are copied into output. After the slab is exhausted (all processed slices are downloaded from slab), filter swaps its phase again to the upload phase and the cycle continues until there are no more slices left.

In the original slice-based streaming implementations, the upload (U) and download (D) phases nicely interleave (except the leading and trailing parts) so that the workload looks as

```
(UUU...)DUDUDU DUD...(DDD...).
```

In BatchStreaming, the two methods interleave in batches, e.g. for batch of size 2 it would be

```
(UUU...)DDUUDDUUDDUUDD...(DDD...).
```

The batch size B represents trade off between two extremes. If $B = 1$ then BatchStreaming turns into a solution very similar to the original streaming solution. On the other side, if $B = nz$ then the BatchStreaming turns into a solution very similar to the AiM one. (In both cases there will be a certain overhead causing that performance will be a little bit worse as in the corresponding compared solution.) Batch streaming so represents the standard "time-space" dilemma available in nearly all optimizations.

Note that the proper batch size B depends on particular algorithm that is implemented with batch streaming.

We implemented separable filtering with the BatchStreaming technique. Similar to the original slice-based streaming, it is advantageous to perform the filtering along the X and Y directions while uploading slices into slab, whereas filtering along the Z direction (i.e. across slices) is done either when downloading from slab or while switching between upload and download phase, when all slices are already in the slab. Due to the mirroring of boundary voxels we use, the leading and trailing slices are of course processed in a special way.

The interesting question is what is the best batch size from the performance point of view? We originally expected that the best batch size for all volumes and filters will be the largest

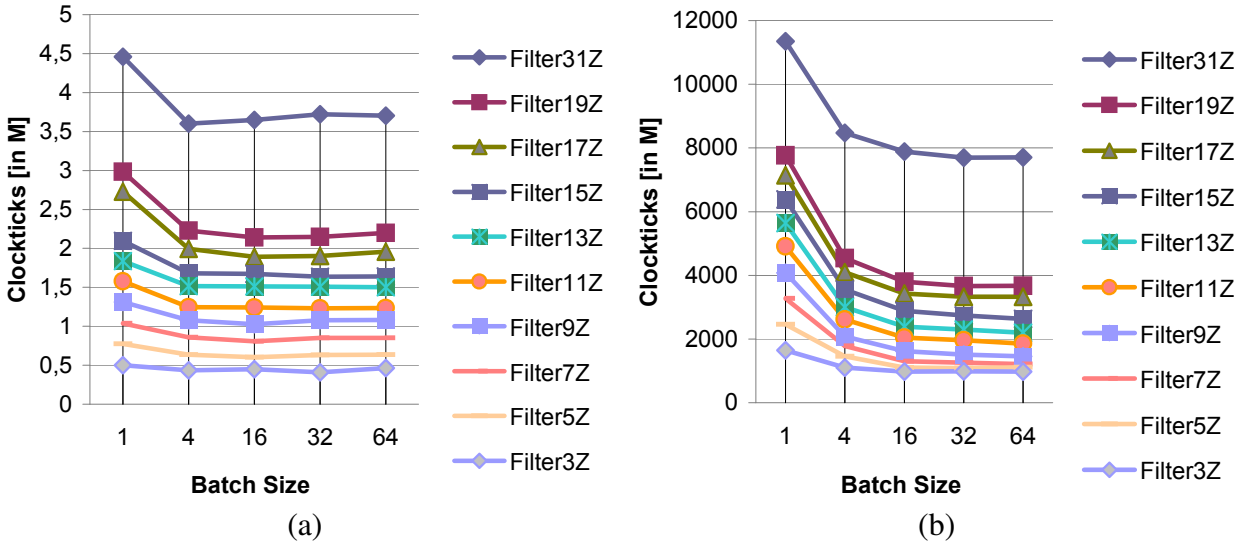

Figure 8: Impact of batch size on filtering of (a) vol0 and (b) vol11 with filters of various sizes.

one, i.e. when as many slices as possible are loaded in the slab. Well, this is not always true. Our tests confirmed this hypothesis only partially. We run many tests in order to understand the performance of separable filtering with BatchStreaming. From the tests we learned that the filtered volumes can be divided intro three categories.

In the first category there are small volumes that fits into the L2 cache:

$$nz \cdot sliceSize \cdot \text{sizeof}(\text{float}) < L2. \tag{8}$$

Figure 8a shows the impact of batch size B on filtering of vol0, which belongs to this first category. It is 1 MB in size and fits nicely into the L2 cache of our computer which is 4 MB. We can see that results for all batch sizes except the first one ($B = 1$) are very similar and better as in the case when $B = 1$. It is also interesting to see that there is no difference in results, no matter which batch size $B \geq 4$ is used. This can be explained by realizing that the performance penalty for an L1 miss followed by an L2 hit is rather low. Additionally, due to small slice strides, the HW prefetchers that prefetch from L2 cache into L1 cache may be often successful and so further decrease this low penalty.

The second category contains huge volumes, such that the slab does not fit into the L2 cache even when $B = 1$:

$$dz \cdot sliceSize \cdot \text{sizeof}(\text{float}) > L2. \tag{9}$$

Impact of batch size on filtering of vol11 is shown in Figure 8b. One slice of vol11 occupies nearly 16 MB in memory, so the slab definitely does not fit into the L2 cache for any filter size. We can see that with increasing batch size the number of clockticks decreases but, for our great surprise, not linearly. Larger filters benefit more from larger batch size as the short filters can do.

Remaining volumes belong to the third category. These volumes do not fit into the L2 cache, but the corresponding slabs (with respect to the filter and for small batch sizes) does.

Clockticks	Batch Size						
[in M]	1	4	16	32	64	128	256
Filter3Z	29.06	25.54	39.93	62.21	71.61	76.44	77.90
Filter5Z	52.00	39.12	65.38	76.53	79.48	80.04	80.94
Filter7Z	69.31	52.65	80.41	83.07	82.05	81.59	82.49
Filter9Z	88.60	67.38	95.91	92.88	89.30	88.04	87.70
Filter11Z	107.96	102.75	110.83	97.98	91.75	88.67	90.26
Filter13Z	153.61	163.85	130.68	110.83	103.51	99.31	101.36
Filter15Z	414.76	226.54	157.29	136.72	128.04	123.17	125.63
Filter17Z	617.57	267.62	166.91	147.59	134.63	129.76	128.48
Filter19Z	775.37	310.63	191.57	174.12	157.23	150.74	151.02
Filter31Z	1482.83	544.50	350.12	315.42	295.78	286.09	281.50

Table 3: Comparing impact of batch size on filtering of vol4. The red numbers represent the case, when the slab (which contains $B + dz - 1$ slices) fits into our L2 cache, i.e. slab must contain fewer than 16 slices.

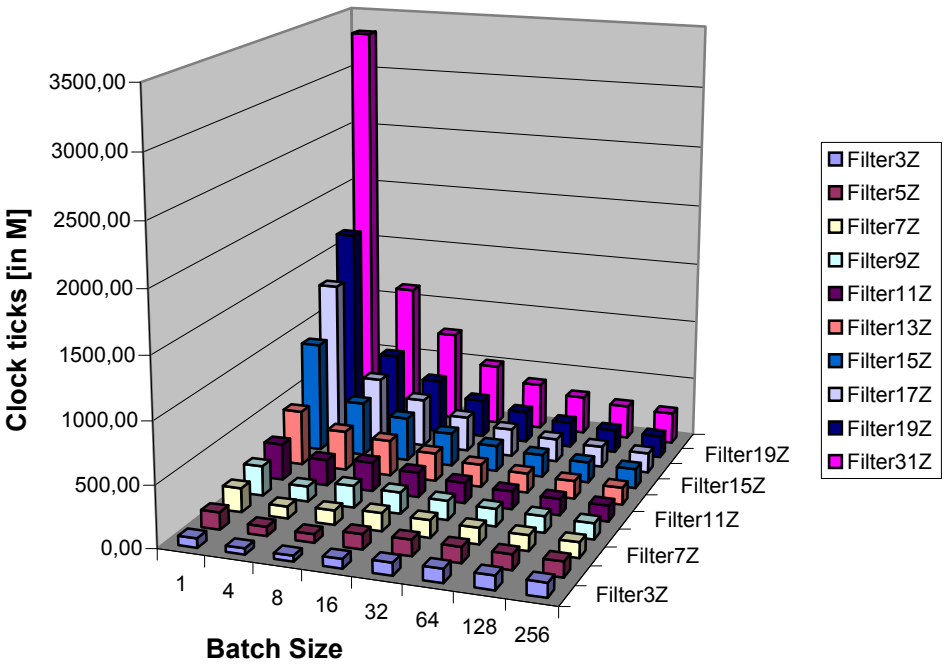

Figure 9: Impact of batch size on filtering of vol4 (third category) with filters of various sizes.

Table 3 and Figure 9 compare batch streaming filtering with various batch sizes and multiple filters on vol4, which belongs to the third category. One slice of volume vol4 is of size $0,25$ MB. Thus all slabs that consist of at most 16 slices will fit in our L2 cache, which is 4 MB in size. Remember that in separable filtering with batch streaming the slab consists of exactly $dz + B - 1$ slices. For vol4 and tested filter sizes, such cases (i.e. when $dz + B - 1 < 16$) are in Table 3 typed in red.

For finding proper batch size value we propose the following (empirically based) algorithm:

1. If the whole volume fits into the L2 cache, then use $B = nz$ (volumes in the first category).
2. If the slab does not fit into the L2 cache even for $B = 1$ (second category of volumes) then:
   (a) If memory is not a problem, find and use the highest B such that the slab will fit into main memory.
   (b) If not only performance but also low memory usage is important, then use batch sizes around 32.
3. If the slab fits into the L2 cache only for certain values of B (third category of volumes), then find and use the highest B such that the slab fits into the L2 cache.

The branch 2b is important if for example we want to run more processes in parallel, so we cannot use the whole main memory for just one process. In this case we are searching for the proper time/space ratio, i.e. the algorithm performs well and still the memory consumption is being kept relatively low. For this purpose we can use batch sizes around 32 which yield good results.

Results

We implemented separable filtering with BatchStreaming in the f3d suite. Additionally, we optimized and vectorized it similar as in AiM.

Figure 10 shows the relative speedups obtained by running the separable filtering of size 7^3 in the 32-bit environment using batch streaming. All speedups are high and are caused by using SSE and additional optimization techniques. Additionally, the speedups further increase with increasing filter size, the naive version is for example 56 times slower than the SSE version, when filtering vol14 with the filter of length 19 in the Z direction. Theses high speedups are, of course, caused mainly by poor performance of the naive version, which does not use any optimizations in memory layout.

If we compare the AiM versions with BatchStreaming we can see that the AiM version is nearly always faster than the batch streaming with properly calculated batch size (in the Z direction). However, there are a few exceptions. For example, for filter of size 7 and volumes vol2 ... vol5, vol12 and vol13 the batch streaming SSE version is faster than the corresponding AiM version. For all of these volumes, the computed batch size ensures that the slab fits into the L2 cache. As the slab in batch streamed filtering is processed right after being filled with slices and

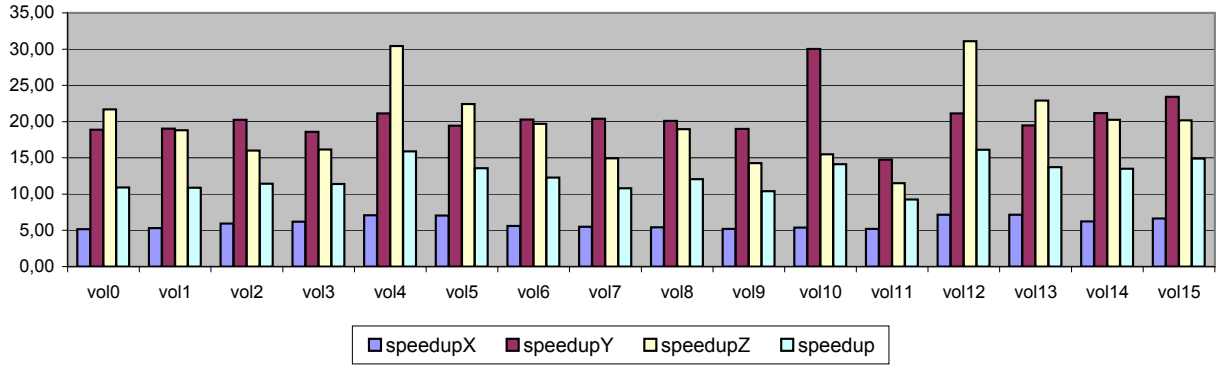

Figure 10: Measured speedups ($SSE|N$) of the separable filter of size 7^3 on the 32-bit architecture using batch streaming.

the slab fits into the L2 cache, the computation in the Z direction is performed faster as in the case of AiM, where all slices are first filtered in the X and Y directions, so the slices are not in the L2 cache when filtering in the Z direction takes place.

Gaussian Filtering

The 3D Gaussian filter is defined by

$$g_{3D}(x,y,z) = \frac{1}{\sqrt{8\pi^3}\sigma_x\sigma_y\sigma_z} e^{-\left(\frac{x^2}{2\sigma_x^2}+\frac{y^2}{2\sigma_y^2}+\frac{z^2}{2\sigma_z^2}\right)}, \tag{10}$$

where σ is the standard deviation of the Gaussian filter and x, y, and z are the voxels coordinates relative to the center of the filter. It samples voxels in a symmetric neighborhood of the input volume and computes their weighted average (convolution). This convolution can be performed more efficiently if we consider that the Gaussian is linearly separable, i.e.

$$g_{3D}(x,y,z) = g_{1D}(x)\cdot g_{1D}(y)\cdot g_{1D}(z) = \\ \frac{1}{\sqrt{2\pi}\sigma_x} e^{-\left(\frac{x^2}{2\sigma_x^2}\right)} \cdot \frac{1}{\sqrt{2\pi}\sigma_y} e^{-\left(\frac{y^2}{2\sigma_y^2}\right)} \cdot \frac{1}{\sqrt{2\pi}\sigma_z} e^{-\left(\frac{z^2}{2\sigma_z^2}\right)}. \tag{11}$$

In theory, the Gaussian filter has an infinite support. However, from a practical point of view its magnitude can be in a distance of 3–4 σ from its center neglected.

Filtering by the Gaussian filter, and filters derived from it, is a common operation often applied to various data classes, including volumetric (three dimensional, 3D) data sets and, eventually, their temporal sequences. On the one hand, Gaussian filtering is a popular standalone tool for smoothing and noise removal, and its spatial derivatives are often used for gradient estimation and edge detection.

On the other hand, filters of the Gaussian family are a common component of other complex operations.

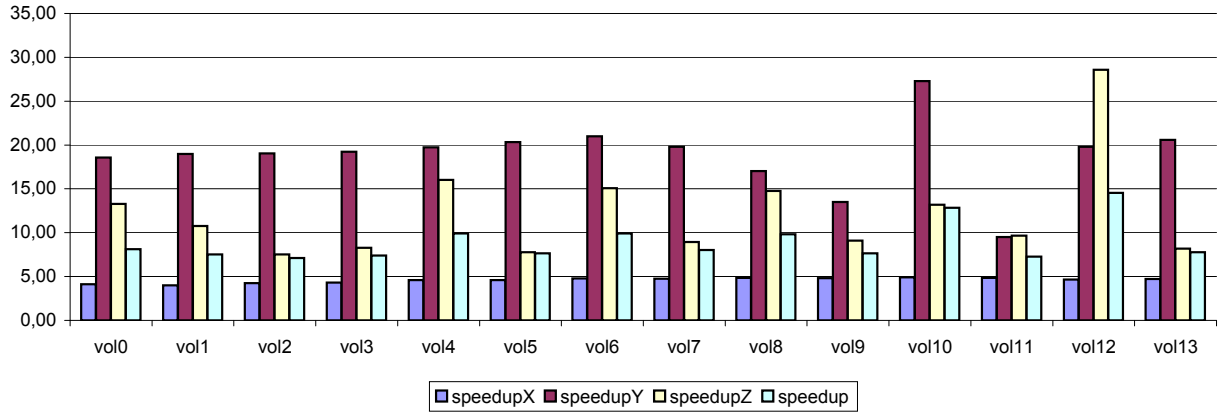

Figure 11: Measured speedups ($SSE|N$) of the recursive Gaussian (all-in-memory version) on the 32-bit architecture.

Due to the popularity and wide applicability of the Gaussian filtering it is very desirable to perform it as fast as possible. As the Gaussian filter is separable and symmetric we can apply it as a separable filter, the optimization which we already described. However, we will now deal with optimization and vectorization of the recursive (IIR) Gaussian implementation first described by Young and van Vliet in [YvV95].

In AiM, the IIR Gaussian can be implemented in two passes. In the first one, forward and backward difference equations are applied in the X and Y directions, whereas in the second pass, difference equations in the Z direction are applied. Things get complicated in streaming where only 4 slices are available in slab. Here, we also process volume in two passes. In the first pass, we apply forward and backward difference equations in the X and Y directions and additionally the forward difference equation in the Z direction. In the second pass, the volume is read in reversed order and only the backward difference equation in the Z direction is applied.

Vectorization of IIR Gaussian is similar to vectorization of separable filtering. However, in the X direction we first have to make the computation of four consecutive voxels mutually independent. After this, we can successfully apply the shuffling approach from symmetric filtering. In the Y direction (in AiM also in the Z direction), a technique similar to the already introduced BlockSize technique may be used. In streaming in the Z direction we use the DirectVectorization approach from separable filtering.

Figure 11 shows the relative speedups obtained by running the AiM of recursive Gaussian in the 32-bit environment. We can see that speedups are high in all directions. In streaming, the speedups in the X and Y directions are very similar, but the speedups in the Z direction are lower (about 2). This is due the fact that in streaming the BlockSize technique cannot be used in the Z direction.

Our implementation of IIR Gaussian can also profit from multiple cores/processors if compiled with OpenMP being enabled. The overall speedup is about 1.5 but only in AiM. In streaming, only the X and Y directions benefit from multiple cores/processors (average speedup is about 1.6).

Detection of Tubular Structures

Now we will deal with optimization of a particular filter used in medical area for detection of tubular structures (vessels for example) [FNVV98]. In order to detect tubular structures over a large size range, multi-scale analysis is performed. This means that the detection filter is executed multiple times. It is therefore desirable that the filter is performed as efficiently as possible.

The filter consists of 3 steps:

1. *Hessian computation.* Hessian is a square matrix of second-order partial derivatives of a function. In the case of volume data, this is a 3×3 matrix. In order to approximate second order derivatives we use the second order finite differences method which turns into evaluation of a $5 \times 5 \times 5$ separable filter.

2. *Computation of Eigenvalues.* Eigenvalues in general are computed using various iterative methods, which are time-intensive [GNU10]. In the case of 3×3 matrix, eigenvalues computation leads to finding roots of a cubic equation. As our matrix is symmetric, we can use the non-iterative numerically stable method introduced in [Ebe11].

3. *Analysis of Eigenvalues.* After eigenvalues have been computed in step 2 they are analyzed to determine locally the likelihood that a tubular structure is present. Thus, for the currently being processed voxel, the output from the filter is the probability that the voxel represents a tubular structure.

The filter operates on floating-point voxels in single-precision. However, due to the numerical stability of the computed result, the eigenvalues computation and their analysis is performed in double-precision. The final result is then converted to single-precision while storing to the output.

In the naive version, all three filter steps are performed on a voxel basis. However, for efficient Hessian computation, it is better to precompute the second order derivatives for the Hessian matrix for the whole slice at once. For this, an additional auxiliary buffer is required. The main reason for this reorganization is that the naive version does not reuse the derivatives computed for the Hessian matrix between single iteration.

Second order derivatives are calculated using second order finite differences. Due to symmetry, it is sufficient to compute only 6 of them which we denote XX, YY, ZZ, XY(=YX), YZ(=ZY) and XZ(=ZX). The derivatives could be computed using multiple $5 \times 5 \times 5$ filters. However, as the coefficients are simple, we can write specialized implementation for calculating derivatives. Due to the nature of the coefficients, this implementation consist only from data shuffling and addition/subtraction (no multiplications are needed).

SSE vectorization of computing of the YY and ZZ and YZ derivatives is straightforward and is similar to the DirectVectorization technique. Vectorization of computing the XX derivative requires data shuffling. The amount of shuffling instructions can be cut in half if we reuse the intermediate shuffled results across the iterations. The remaining two derivatives, XY and XZ, also require quite a lot of shuffling. However, this can be greatly reused if we first compute the first order derivatives in the X direction and compute the second order derivatives XY and XZ derivatives as first order derivatives from the already computed first order derivative in the X direction.

With other words, we replace the direct computation of two second order derivatives (XY and XZ) from input data with computing three first order derivatives, the first one (in the X direction) only being used as input for computing XY and XZ derivatives as first order derivatives.

In order to compute eigenvalues we optimized and vectorized the algorithm from [Ebe11]. Due to the numerical stability, the eigenvalues are computed in double-precision using SSE2. When we iterate through the slice, we process four voxels at once. However, SSE2 can operate only on two double-precision values in parallel, eigenvalues must be therefore computed in two steps. Additionally, the derivatives are computed in single-precision and must be converted to double-precision for eigenvalues computation. This conversion is performed on demand when the eigenvalues are computed.

The vectorization of computing eigenvalues is straightforward. First, we slightly optimized the computation so it uses less multiplications and different variables. Next, we rewrote the algorithm in a branch-free manner. Additionally, as eigenvalues computation of 3×3 matrix leads to finding roots of a cubic equation, efficient SSE2 implementation of trigonometric and cyclometric functions is required. Although it is known that SSE2 can speedup the computation of trigonometric functions [NNS08], we did not succeed in finding a good free library that would offer this functionality. Therefore, we had to implement an efficient SSE2 version of the required trigonometric and cyclometric functions.

Similar to vectorization of eigenvalues computation, vectorization of analyzing eigenvalues using the vesselness function was straightforward, after we rewrote it into a branch-free version. Additionally, we had to vectorize the `exp` function from the standard library, so that is uses SSE2.

From eigenvalues analysis we finally obtain four double values that represent the probability of the corresponding voxel being a tubular structure. This four probabilities are converted into single-precision floating points and stored to the output.

As the eigenvalues computation and analysis is independent, we parallelized it using OpenMP, so multiple rows of a slice can be processed in parallel.

We implemented the tubular structures detection filter in the f3dfilter library which is part of the f3d suite ([SD02, VVS+07]). There are three versions of the detection filter: naive (denoted N), GSL version and a version called SSE. The naive (N) version, as the name suggests, does not use any SIMD optimizations. The eigenvalues are computed using the non-iterative algorithm and on a per voxel basis. The GSL version is similar to the naive version, however, it uses the GSL[GNU10] library for computing eigenvalues. The SSE version is fully optimized for SSE and SSE2. Additionally, it precomputes second order derivatives (on a slice basis) from which the eigenvalues are then computed using the non-iterative algorithm.

Table 4 shows results for the 32-bit version. We can see, that the naive (N) version is 1.37 – 1.47 times faster than the GSL version. The reason lies in eigenvalue computation, as the GSL versions uses the iterative approach for this. The SSE/SSE2 version achieves also nice speedups. When the slab, which consists of 5 slices, fits into the L2 cache (4 MB in our case) then we measured the speedup above 13. If the slab does not fit into the L2 cache, the speedup drops to 4, which is slightly better as we could expect from SSE optimization.

As enabling OpenMP did not significantly improve the performance of detection filter, further improvements in this area should be subject of future work.

	Clockticks [in M]			Speedup	
	N	GSL	SSE	$N\|GSL$	$SSE\|N$
vol0	106.0	155.1	7.9	1.46	13.37
vol1	205.7	301.9	15.1	1.47	13.62
vol2	836.1	1222.2	62.2	1.46	13.44
vol3	1186.4	1737.3	86.9	1.46	13.65
vol4	7358.9	10493.4	532.3	1.43	13.83
vol5	5487.8	8049.9	404.8	1.47	13.56
vol6	54286.0	79617.9	12497.5	1.47	4.34
vol7	58983.9	86792.5	13726.2	1.47	4.30
vol8	127164.5	174883.3	26168.9	1.38	4.86
vol9	104149.1	152918.3	25262.2	1.47	4.12
vol10	127830.8	175155.3	25957.8	1.37	4.93
vol11	79806.1	116878.7	19178.8	1.46	4.16
vol12	58857.8	84238.3	4268.2	1.43	13.79
vol13	46515.9	68154.8	3427.7	1.46	13.57
vol14	509040.3	699613.8	104619.7	1.37	4.87
vol15	2220158.5	3035470.3	450393.4	1.37	4.93

Table 4: Results of tubular structures detection filter (32-bit architecture).

Conclusion and Future Work

In this thesis we devoted to optimizing filtering (convolution) of volumetric data in two different environments – AiM (all-in-memory) and streamed. We analyzed both approaches from the point of view of performance and proposed new solutions which allowed significant speedup. Namely we introduced new memory layout for efficient processing of volumetric data which allow usage of more efficient loop patterns. In streaming we designed a new slices workflow. A special care was devoted to optimization of Gaussian filtering, which is widely used in many applications. Additionally, we optimized a concrete filter for detecting tubular structures. All filtering algorithms were vectorized for SSE on assembler level. We wrote many assembler routines which are specific to the underlying hardware architecture (32-bit, 64-bit) and operating system (Windows, Linux). All our algorithms were designed (and implemented) in such way that they are easily parallelized with the help of OpenMP and can make use of multiple processors or processors with multiple-cores, which are nowadays common in our desktop processors.

For achieving best performance and squeezing the most from the CPU, some of our algorithms require special configuration with respect to the hardware they are currently running on. For simplicity, we implemented special routines so that our algorithms are automatically self-configured in runtime and the user must not care about the configuration.

We fulfilled the aims of thesis – the algorithm of filtering of volumetric data was redesigned so that it runs efficiently also on the newest desktop computers. However, the algorithm optimization is a never-ending process. It requires to analyze bottlenecks and to design and implement

new solutions. The new solutions are again subject of analysis for another bottlenecks and so on. So it is with our work. There are some issues that left for the future. First, the application of OpenMP should be further tested whether our implementations scale nice with increasing number of cores/processors. As we were primarily concerned in algorithm optimization and vectorization we think that here left room for another enhancements. Second, we always filtered volumes in single-precision. However, in some areas of computing, the double-precision floating-point calculation are required. This would require to write special implementations using SSE2 instead of using SSE. Finally, in this year (2011) we will be witnesses of another milestone in processor evolution. The 128-bit SSE instruction set will be extended to 256-bit one called AVX. As volume filtering is a floating point intensive calculation, it could benefit from AVX for sure.

Filtering of volumetric data is used in many complex algorithms. Therefore, highly optimized and tuned filtering will also optimize these complex algorithms. We hope that optimization techniques described in this thesis can be applied to another computational intensive tasks and so will bring in new possibilities to the area of volumetric data processing.

Bibliography

[BDG+97] David Bistry, Carole Dulong, Mickey Gutman, Mike Julier, Michael Kieth, Lawrence M. Mennemeier, Millind Mittal, Alex D. Peleg, and Uri Weiser. *The Complete Guide to MMX Technology*. Mc Graw Hill, third edition, 1997.

[BO03] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, first edition, 2003.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[Ebe11] David Eberly. Eigensystems for 3×3 Symmetric Matrices (Revisited). Technical report, Geometric Tools, LLC, February 2011. [Last accessed 26.2.2011].

[FNVV98] Alejandro F. Frangi, Wiro J. Niessen, Koen L. Vincken, and Max A. Viergever. Multiscale vessel enhancement filtering. In *MICCAI'98 - Medical Image Computing and Computer-Aided Intervention*, pages 130 – 137, 1998.

[GNU10] GNU. GSL - GNU Scientific Library [Internet]. `http://www.gnu.org/software/gsl/`, March 2010. [Last accessed 26.2.2011].

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23:5–48, March 1991.

[Int99] Intel. Real and Complex FIR Filter Using Streaming SIMD Extensions, order number: 243643-002. `www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/245711.htm?page=1`, 1999. [Last accessed 23.6.2009].

[Int10] Intel. Intel©64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Order Number: 253668. `http://www.intel.com/Assets/PDF/manual/253668.pdf`, 2010. [Last accessed 16.10.2010].

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[NNS08] Lars Nyland, Lars Nyland, and Mark Snyder. Fast Trigonometric Functions Using Intel's SSE2 Instructions. *CiteSeerX - Scientific Literature Digital Library and Search Engine [http://citeseerx.ist.psu.edu/oai2] (United States)*, 2008. [Last accessed 26.2.2011].

[SD02] M. Sramek and L. I. Dimitrov. `f3d`—a file format and tools for storage and manipulation of volumetric data sets. In G. M. Cortelazzo and C. Guerra, editors, *1st International Symposium on 3D Data Processing, Visualization and Transmission*, pages 368–371. IEEE Computer Society, Padova, Italy, June 2002.

[VVS+07] Andrej Varchola, Anton Vaško, Viliam Solčány, Leonid I. Dimitrov, and Miloš Šrámek. Processing of volumetric data by slice- and process-based streaming. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 101–110, New York, NY, USA, 2007. ACM.

[YvV95] I.T. Young and Lucas J. van Vliet. Recursive implementation of the Gaussian filter. *Signal Processing*, 44:139–151(13), June 1995.

Scientific Activities

Publications

Anton Vaško and Miloš Šrámek. Optimizing Gaussian filtering of volumetric data using SSE. *Concurrency and Computation: Practice and Experience*, 23(1):100–116, 2011.

Andrej Varchola, Anton Vaško, Viliam Solčány, Leonid I. Dimitrov, and Miloš Šrámek. Processing of volumetric data by slice- and process-based streaming. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 101–110, New York, NY, USA, 2007. ACM.

Theses

Anton Vaško. *Efficient Volume Filtering*. Project of Dissertation, FMFI UK, Bratislava, 2007.

Anton Vaško. *SIMD Optimization in Volume Rendering and Gaussian Filtering*. Rigorous Thesis, FMFI UK, Bratislava, 2007

Anton Vaško. *SIMD Optimization in Volume Rendering*. Master's Thesis, FMFI UK, Bratislava, 2005

Research projects

Tools for processing and visualization of tomographic and confocal data , 2006 – 2009, APVV, Slovakia, No. APVV-20-056105