

# Acceleration of Multiresolution Imaging Algorithms: A Comparative Study

Richard Membarth, Philipp Kutzer, Hritam Dutta, Frank Hannig, and Jürgen Teich  
 Hardware/Software Co-Design, Department of Computer Science  
 University of Erlangen-Nuremberg, Germany  
 Email: {richard.membarth, dutta, hannig, teich}@cs.fau.de, philipp.kutzer@stud.fau.de

**Abstract**—In this paper we consider a multiresolution filter and its realization on the Cell BE and GPUs. We not only present common and specific optimization strategies undertaken for obtaining maximum performance on these architectures, but also how to obtain a speedup of 6.57x and 33.24x compared to an optimized OpenMP baseline implementation. Furthermore, we also undertake automated configuration space exploration of different partitioning possibilities for selection of best tiling parameters.

## I. INTRODUCTION

Characteristic for the class of multiresolution algorithms is to process a signal on different resolutions. The field of applications for multilevel algorithms is diverse. In the JPEG 2000 and MPEG-4 standards, the discrete wavelet transform, which is also a multiresolution approach, is used for image compression. In medical imaging, multiresolution filters offer much better edge-preserving smoothing for preserving the visually important structures [1].

Not only the domain of these algorithms is manifold, but also the platforms that execute the algorithms are diverse. Depending on the application, hardware ranging from standard multi-core systems for MPEG-4 to specialized solutions for medical imaging like DSPs or FPGAs are used. In particular, the parallel nature of the involved algorithms brings new architectures into focus like the Cell Broadband Engine Architecture (CBEA) or graphics processing units (GPUs). The efficient use of parallel architectures requires, however, careful and architecture-specific tuning to implement algorithms. In the majority of cases, the programmer has to optimize the code manually.

An important task in parallel computing is to distribute the work to the available processing elements in order to leverage the available resources. To achieve this, different approaches exist like pipelining of several subtasks or partitioning of the work into tiles. In this paper, we will focus on the latter one.

A lot of work has been done in porting applications to this architectures. Ryoo et al. [2] present a performance evaluation of various algorithm implementations on the GeForce 8800 GTX. Williams et al. [3] evaluated scientific kernel implementations for the Cell. However, their optimization strategy is limited to well known compute-bound tasks.

In this paper, the application of multiresolution filtering is considered. This application is mapped and optimized for two highly parallel architectures, namely the Cell processor and GPUs. Compiler based optimizations as well as manual optimizations are used to exploit each hardware platform. In order to determine the best tiling for the different architectures,

the tiling configuration is investigated and applied to the application. As baseline implementation, we use a parallelized multi-core implementation to compare the acceleration of each implementation.

The major contributions of this paper are threefold: First, we present specific optimizations which are important for each platform in order to map the algorithm efficiently to the hardware. Second, we show that the best tiling configuration is not obvious on each platform. Third, we show that we achieve a quite noticeable acceleration of the multiresolution application.

## II. ALGORITHM

The multiresolution application considered in this paper utilizes a multiresolution approach [1] and employs a bilateral filter [4] as filter kernel. The filter reduces noise significantly, while sharp image details are preserved.

Figure 1 shows the used multiresolution application: In the decompose phase, two image pyramids with subsequently reduced resolutions are constructed. While the images of the first pyramid ( $g_x$ ) are used to construct the image of the next layer, the second pyramid ( $l_x$ ) represents the edges in the image. The actual algorithm of the application is working in the filter phase on  $l_x$ . After the main filter has processed these images, the output image is reconstructed again.

As filter we use a bilateral filter (cf. [4]), which replaces each pixel by an average of geometric nearby (*closeness*) and photometric similar (*similarity*) pixel values as seen in Algo. 1. Only pixels within the neighborhood  $\sigma_d$  of the relevant pixel are considered. The parameter  $\sigma_r$  determines the amount of combination. Compared to the memory access

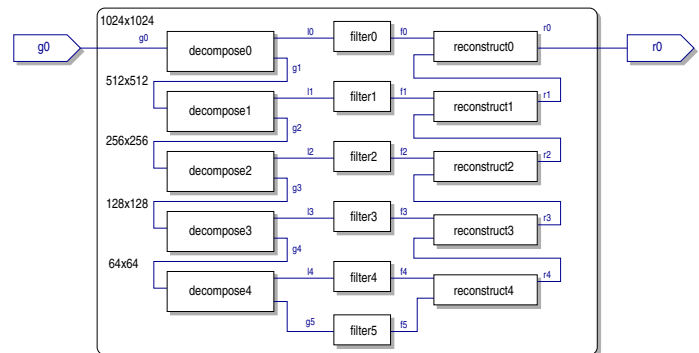


Fig. 1. Multiresolution filter application with five layers.

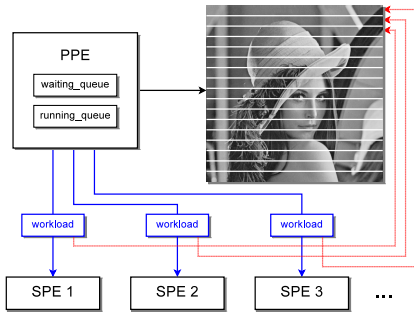


Fig. 2. Partitioning of the image for processing on the Cell: The PPE schedules the workload to available SPEs.

dominating decompose and reconstruct phases, the bilateral filter is compute-intensive.

**Algorithm 1** Bilateral filter used in the multiresolution filter.

```

1: for (yf = -σd; yf ≤ σd; yf++) do
2:   for (xf = -σd; xf ≤ σd; xf++) do
3:     diff = lx[x + xf][y + yf] - lx[x][y]
4:     closeness = expf(-cd * ((xf + σd)2 + (yf + σd)2))
5:     similarity = expf(-cr * diff2)
6:     k += closeness * similarity
7:     p += closeness * similarity * lx[x + xf][y + yf]
8:   end for
9: end for
10: fx[x][y] = p/k

```

III. MAPPING AND OPTIMIZATION STRATEGIES

A. Cell Broadband Engine

The CBEA is a heterogeneous multi-core processor with a theoretical peak performance of 256 GFLOPS [5]. In addition to the main PowerPC Processor Element (PPE) the CBEA possesses 8 Synergistic Processor Elements (SPEs). The SPEs are 128-bit RISC processors, each with a Single Instruction, Multiple Data (SIMD) extension and a 256 KB embedded memory for instruction and data (*local store*).

In order to obtain best performance on the CBEA, it is necessary to exploit the unique features of the hardware like vectorization to operate on 128-bits in one clock cycle. In certain cases, this could be done by a compiler, but in most cases like in image processing it is required to vectorize manually. To access data from main memory, the desired memory area must be transferred by hand per DMA into the local store first. To hide the memory latency, double or multi-buffering can be used.

To partition the computational problem and to offload it to the available processors different approaches like pipelining or tiling are possible. In our application we use the latter one. Therefore, the pictures is divided into blocks which fit into the local store. The PPE schedules these blocks to free SPEs as seen in Fig. 2 and sends corresponding workload packages to SPEs in the waiting queue.

The different layers of the decompose and reconstruct phases are scheduled according to their data dependencies, while filtering is done concurrently on all layers without any synchronization.

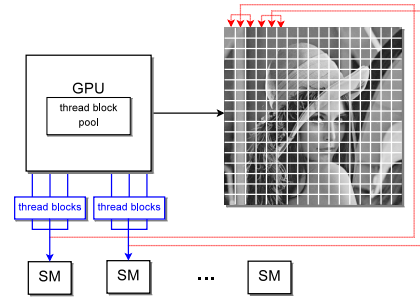


Fig. 3. Partitioning of the image for processing on the GPU: The GPU schedules the thread blocks to available streaming multiprocessors.

B. Graphics Processing Unit

Graphics cards feature hundreds of streaming processors nowadays with a peak performance in the region of TFLOPS, which can be programmed in a C language using the Compute Unified Device Architecture (CUDA) [6] for NVIDIA GPUs. The underlying graphics hardware has several multiprocessors, which have access to the global memory of the GPU as well as to on-chip shared memory. This shared memory can be used by the eight streaming processors of one multiprocessor as a fast scratchpad memory. For extensive transcendental operations there are also special function units available.

To fed the processors steadily with data, contiguous elements in global memory have to be accessed by the streaming processors, so that the data accesses can be grouped into one data transfer. In case the data access is uncoalesced, texture units are used to load data from global memory. This texture memory is not subject to coalescing and is in addition cached. If the same data is used by different streaming processors, the scratchpad is used to stage the access.

A program executed on the GPU is called *kernel* and is processed by many *threads* in parallel on the streaming processors. On the GPU, for each task one separate kernel has to be used, i.e. one kernel for each layer of the decompose, filter, and reconstruct phase.

When a kernel is executed, the image is partitioned into blocks which are processed by the multiprocessors as depicted in Fig. 3.

IV. CONFIGURATION SPACE EXPLORATION

After the programs have been optimized to leverage the unique features of each hardware platform, still the question arises if the hardware resources are best utilized. Therefore, we explore the different configuration options in order to achieve best performance.

A. Cell Broadband Engine

We assumed that the best configuration would be the one, which tiles the complete workload evenly under the 6 SPU of the Cell processor in the PLAYSTATION 3 (PS3). For the first layer of the bilateral filter with a resolution of 1024 × 1024, this leads to an optimal partitioning of 171 (1024/6) lines per block. This can be seen in Fig. 4a showing the execution time for different blocksizes of the bilateral filter for an image of size 1024 × 1024. The minima of the execution time are always those configurations where each SPU processes the same fraction of the image. This is the case for blocksizes of

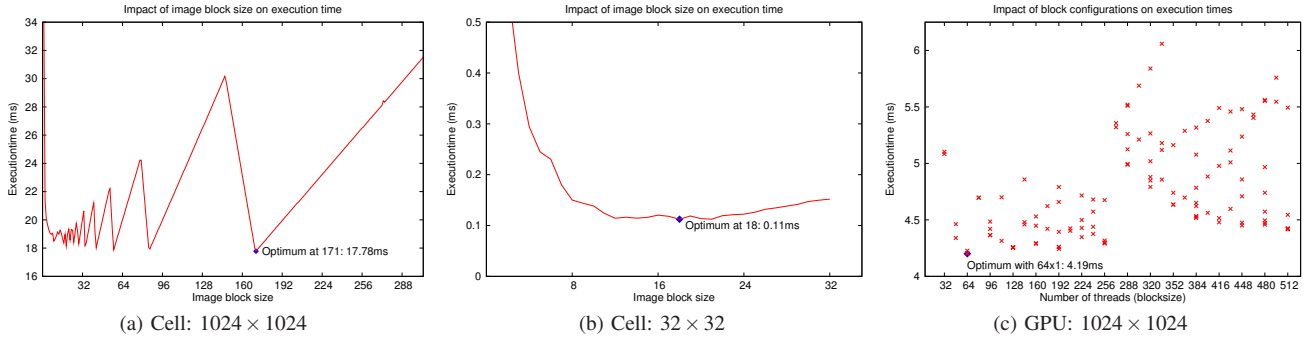


Fig. 4. Configuration space exploration for the bilateral filter ( $\sigma_d = 2$ ) on the PS3 and Tesla C870.

171, 86, 57, 43, etc., where each SPU processes exactly once one sixth of the image, twice one sixth of the image, etc.

However, the best configuration is not obvious anymore as soon as smaller images are considered. For the lowest layer of the filter with a resolution of  $32 \times 32$ , the best configuration has a blocksize of 18 for the workload. This can be seen in Fig. 4b, where only two of the SPUs are used. Although the workload is not evenly distributed, this yields here a better performance and is over 50% faster compared to the configuration with optimal load balancing.

The reason for this is the additional scheduling overhead for small images. The synchronization and communication overhead using all SPUs is too big for small work packages.

### B. Graphics Processing Unit

For the graphics card, the configuration of a kernel has to be specified manually by the programmer. The block configuration determines the number of threads in a block, the shape of the block, and eventually also the resource utilization.

For the 2D block configuration used here, both dimensions can vary between 1 and 512, but the product of them must be equal to or less than 512. That is to say, we have 3280 possible configurations. Since the shape of the block represents usually also the access pattern to the global memory, we can constrain the configurations to be considered. Only the block configurations where the x-dimension is a multiple of 16 are used in order to get coalesced memory access. This limits the number of configurations to 119.

From these configurations, we assumed that a square block with  $16 \times 16$  threads would yield the best performance, utilizing the texture cache best when loading data. However, the configuration space exploration (see Fig. 4c) shows that the best configurations has 64 threads for the x-dimension and 1 thread in the y-dimension, and is hence an 1D configuration. The data set is plotted in 2D for better visualization. Plotted against the x-axis are the number of threads of the block. That is, the configuration  $16 \times 16$  and  $32 \times 8$  have for instance the same x-value. The best configuration is 10.3% faster than the previously as optimal assumed configuration.

## V. EXPERIMENTAL RESULTS

In this section, the architecture specific and architecture independent optimizations are explained. For each technique, the effectiveness for the Cell BE and GPU is analyzed and discussed.

### A. Architecture Specific Optimizations

*Cell Broadband Engine:* We use SIMD instructions to vectorize our code in each step and layer of the algorithm. For a picture of  $1024 \times 1024$ , the algorithm is accelerated by a factor of  $4.60x$ . On the one hand, this speedup is based on vector processing, and on the other hand from less branches as a consequence of the vectorization.

The Cell processor has no built-in branch predictor to decide whether a conditional branch is likely to be taken or not. If a branch is taken, a stall of 18–19 cycles occurs. To avoid this, we use static branch prediction for the border treatment, which reduces the execution time of the complete application by a factor of  $1.08x$  for an image of  $512 \times 512$ .

Double or multi-buffering is used to overlap the DMA transfers and the computations to hide the long DMA latency. However, for our implementation, double buffering yields only a marginal performance gain. Due to a high ratio of processing to waiting time, the time spent on data transfer is negligible small for big images.

*Graphics Processing Unit:* Graphics cards provide texturing units as an alternative to the normal global memory access. In particular, when the memory access is not coalesced, texture memory is used to get better performance. For instance, the decompose kernel is accelerated by a factor of  $3.26x$  and the reconstruct kernel by a factor of  $2.69x$ . The computationally intensive filter kernel has only a speedup of  $1.67x$ .

The computationally intensive bilateral filter uses many multiplications, divisions, and exponential functions, which take longer to execute than other operations. To accelerate these functions, we use intrinsics which execute in less clock cycles, often at the expense of accuracy. Doing so accelerates the implementation of the bilateral filter by a factor of  $1.51x$ .

### B. Architecture Independent Optimizations

*Lookup Table:* With the use of lookup tables instead of the extensive exponential calculations for the similarity and closeness functions, the bilateral filter can be accelerated significantly. Figure 5a shows the relative execution time using lookup tables against the naïve implementation on the Cell. The speedup for the similarity function is twice the speedup of the closeness function since we use shorts to represent pixels and hence, need twice as many exponentiations for the similarity function. The combination of both lookup tables reduces the execution time of the filter kernel by a factor

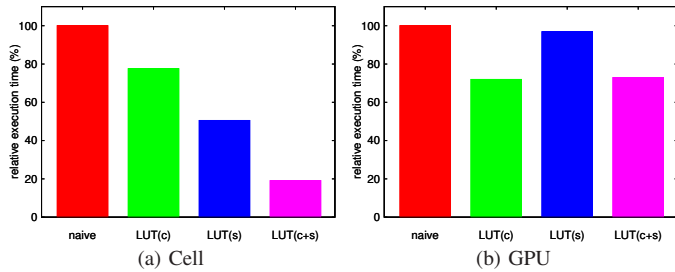


Fig. 5. Relative improvements of the execution times using lookup tables for the closeness and similarity functions.

of  $5.24x$ . Figure 5b shows the corresponding results for the GPU. While the lookup table for the closeness function accelerates the filter by a factor of  $1.39x$ , the lookup table for the similarity function has no further improvements. For the closeness function, all threads access the same element of the lookup table, whereas the memory access pattern for the similarity function depends on the pixel values and is irregular.

*Loop Unrolling:* Unrolling both loops of the bilateral filter for the Cell results in a speedup of  $1.26x$ , while the GPU implementation has a speedup of  $0.89x$ , i.e. the execution takes longer. This is the result of increased register usage of the kernel when both loops are unrolled. When only the inner or outer loop is unrolled, the register usage stays approximately the same and an acceleration by a factor of  $1.02x$  is achieved.

*Loop Fusion:* This optimization is used for the tasks in the decompose and reconstruct phases, which are memory-bound. Initially, for each task of these phases a separate kernel is used. Subsequently, these kernels are merged as long as data dependencies are met. For the Cell, loop fusion accelerates the decompose phase by a factor of  $5.17x$  and the reconstruct phase by  $3.58x$ . Similarly, the GPU implementations are accelerated by a factor of  $2.35x$  and  $2.28x$ , respectively.

TABLE I  
SUMMARY OF ARCHITECTURE SPECIFIC AS WELL AS ARCHITECTURE INDEPENDENT OPTIMIZATIONS.

	Cell	GPU
SIMD	$4.60x$	–
double buffering	$1.00x$	–
branch prediction	$1.08x$	–
texture memory	–	$3.26x$
intrinsic	–	$1.51x$
lookup table	$5.24x$	$1.39x$
loop unrolling	$1.26x$	$1.02x$
loop fusion	$5.17x$	$2.35x$

### C. Comparison

*Optimizations:* Table I summarizes the optimizations applied on each platform. Considering the architecture specific optimizations, it can be seen that vectorization has the biggest impact for the Cell, while double buffering and static branch prediction have only a marginal impact. For GPUs, in particular the usage of texture memory improves the performance, but also the intrinsics have some impact. For the architecture

independent optimizations, lookup tables and loop fusion lead to a remarkable speedup.

*Performance:* Finally, we compare the performance our optimized Cell and GPU implementations achieve with a parallel baseline implementation on a Xeon Quad Core E5430 with 2.66 GHz using OpenMP to leverage all available cores. Table II compares the performance achieved on a Tesla C870 and a PS3 with the parallelized Xeon implementation. It can be seen that both implementations achieve a remarkable speedup. While the PS3 obtains a speedup of about  $6.57x$  and can process images up to  $1024 \times 1024$  in real-time, the Tesla C870 outperforms the Xeon even by a factor of  $33.24x$  for an image of  $2048 \times 2048$  (36.17 FPS, not shown in the table).

TABLE II  
SPEEDUP AND FPS FOR THE MULTIREOLUTION FILTER ON TESLA C870 AND THE PS3 COMPARED TO A XEON QUAD CORE (2.66 GHz) DIFFERENT IMAGE SIZES ( $\sigma_d = 2$ ).

	$512 \times 512$		$1024 \times 1024$	
	FPS	Speedup	FPS	Speedup
Xeon	17.29	–	4.64	–
Cell	109.29	6.32	30.50	6.57
Tesla	382.13	22.10	130.99	28.23

## VI. CONCLUSION

The idiosyncrasies of diverse multi-core architecture require careful consideration of compiler optimization effects. In our work, we show the significant effect of several architecture dependent optimizations like vectorization, etc. and independent optimizations like loop fusion and loop unrolling on the execution time. Their synergistic implementation leads to a speedup of  $6.57x$  and  $33.24x$  for Cell BE and GPU, respectively against an optimized OpenMP implementation. The optimal tiling parameters for load partitioning onto these architectures have relevant influence on latency and therefore need to be correctly determined. The empirically selected parameters have 10.3% better performance on the GPU than obvious model-based ones and up to 50% better performance on the Cell BE for small images.

## REFERENCES

- [1] D. Kunz, K. Eck, H. Fillbrandt, and T. Aach, “Nonlinear Multiresolution Gradient Adaptive Filter for Medical Images,” in *Proceedings of the SPIE: Medical Imaging 2003: Image Processing*, vol. 5032, San Diego, CA, USA, 2003, pp. 732–742.
- [2] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Wen-Mei, “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, Salt Lake City, UT, USA, 2008, pp. 73–82.
- [3] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, “The Potential of the Cell Processor for Scientific Computing,” in *Proceedings of the 3rd Conference on Computing Frontiers*, ACM New York, NY, USA, 2006, pp. 9–20.
- [4] C. Tomasi and R. Manduchi, “Bilateral Filtering for Gray and Color Images,” *Proceedings of the Sixth International Conference on Computer Vision*, pp. 839–846, 1998.
- [5] M. Gschwind, “Chip Multiprocessing and the Cell Broadband Engine,” in *Proceedings of the 3rd conference on Computing Frontiers*, ACM New York, NY, USA, 2006, pp. 1–8.
- [6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” *IEEE Micro*, pp. 39–55, 2008.