Run-time Adaptation to Heterogeneous Processing Units for Real-time Stereo Vision

Benjamin Ranft
FZI Research Center for Information Technology
Karlsruhe, Germany
ranft@fzi.de

Oliver Denninger
FZI Research Center for Information Technology
Karlsruhe, Germany
denninger@fzi.de

***Abstract*—Todays systems from smartphones to workstations are becoming increasingly parallel and heterogeneous: Processing units not only consist of more and more identical cores – furthermore, systems commonly contain either a discrete general-purpose GPU alongside with their CPU or even integrate both on a single chip. To benefit from this trend, software should utilize all available resources and adapt to varying configurations, including different CPU and GPU performance or competing processes.**

This paper investigates parallelization and adaptation strategies using dense stereo vision as an example application – a basis e. g. for advanced driver assistance systems, but also robotics or gesture recognition. At this, task-driven as well as data element- and data flow-driven parallelization approaches are feasible. To achieve real-time performance, we first utilize data element-parallelism individually on each processing unit. On this basis, we develop and implement further strategies for heterogeneous systems and automatic adaptation to the hardware available at run-time. Each approach is described concerning i. a. the propagation of data to processors and its relation to established methods. An experimental evaluation with multiple test systems and usage scenarious reveals advantages and limitations of each strategy.

***Keywords*-multicore processing; parallel programming; scheduling algorithm; image processing**

I. Introduction

When capturing images with a standard camera, depth information of the scene is in most cases lost. Stereo vision is the process of recovering 3D structure from images of two side-by-side cameras, making it an important basis for environmental perception. Its applications include emergency braking assistants [1] or human-machine interfaces [2], both of which plausibly require real-time performance. However, it is computationally demanding to estimate a distance for ideally every pixel of high resolution images.

Serial or badly scaling implementations of stereo vision are not likely to benefit much from current or future processors, because power consumption and memory bandwidth limit the latters' clock frequencies [3]. Instead, applications must make use of the continuosly growing number of parallel processing units. Fortunately, image processing – including most stereo vision algorithms – is usually well-suited for large-scale data element-parallelism: In the most simple case, identical operations can be carried out for each pixel independently. For such workloads graphics processing units (GPUs) often outperform multi-core CPUs with single instruction, multiple data (SIMD) capabilities w. r. t. frame rates and energy efficiency [4], [5], although there are exceptions [6]. CPUs in contrast stand out at sequential or heavily branching tasks, but are also generally required to run an operating system and control GPUs. Consequently heterogeneous systems offering several CPU cores together with one or more GPUs as co-processors will probably become increasingly common. This requires computationally intensive applications to:

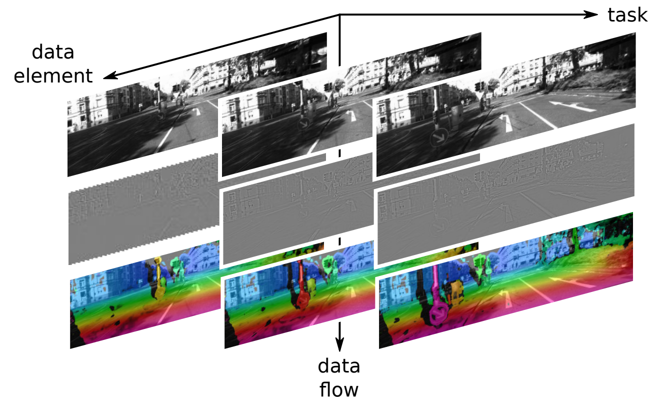

Figure 1. Parallelization approaches offered by our application

- efficiently utilize different systems by supporting multiple CPU cores, SIMD capabilities and GPUs, and by implementing algorithms suitable for each architecture
- take into account the memory model of a heterogeneous system, which – in contrast to shared memory systems – may resemble distributed systems in terms of increased coordination and data transfer efforts
- dynamically adapt to varying target system performance due to different hardware, algorithm characteristics and competing processes, since it would be hard and inflexible to statically define or pre-compute adequate scheduling schemes

This paper extends the work of [4] – its contribution consists of the realization of especially the third above-named aspect. Its remainder is organized as follows: Related work w. r. t. parallelization and adaptation approaches as well as comparable stereo vision implementations are presented in section I.A, while section I.B describes the hardware

architectures and programming interfaces used in our test systems. Section II will summarize the algorithms and the device-specific data element-parallelism of our stereo vision implementation. On this basis, section III introduces different strategies to simultaneously utilize and adapt to multiple heterogeneous processing units, and relates them to our application's parallelization approaches shown in Figure 1. Section IV conducts an experimental evaluation of the proposed approaches. Section V concludes the paper and gives an outlook to future works.

A. Related Work

Even before the trend towards GPU computing, mainly video games were able to distribute the task of 3D rendering across 2 - 4 GPUs. Today's corresponding implementations offer two common modes of operation [7]: At *split frame rendering* each GPU processes a partition of every frame whose size is determined by the complexity of its content. In contrast, *alternate frame rendering* assigns each frame to one of the GPUs via round-robin, which requires cooperating devices to be very similar or identical for efficient operation.

Beyond this wide-spread yet specific application, there has been notable research towards cooperatively using heterogeneous processing units for general-purpose computations: [8] determines an optimal workload distribution ratio between CPU and GPU during initialization, while [9] includes multiple different GPUs and exploits its task's tree-like dependency structure to increase efficiency – both confirm the importance of adapting to the target system at runtime. More generally, programming heterogeneous hardware can be simplified by unifying the interface to architecture-specific code and automatically handling inter-device data-transfers [10], or by enabling applications intended specifically for *nVidia* GPUs to run on those by *AMD* and on multi-core *x86*-CPUs as well [11].

Concerning stereo vision as our sample application, available open-source implementations differ w. r. t. both algorithms and parallelization: [12] reduces complexity via a sparse prior on depth information and uses SIMD instructions of *x86*-CPUs. Despite the lack of any parallelization, [13] has inspired our CPU implementation because of its computational efficiency. An early and well-optimized GPU implementation of stereo vision was presented by [14]. However, we found the *OpenCV* library [15] to be the most suitable benchmark for this paper: If built accordingly, it extensively uses SIMD instructions and supports multiple CPU cores via *Intel Threading Building Blocks* as well as GPUs via *nVidia C for CUDA*. Figure 2 shows typical results of its stereo vision algorithms, while Table I lists their respective frame rates on our test systems[1]. Our algorithms[2] are comparable to the C++ classes *cv::StereoBM* and

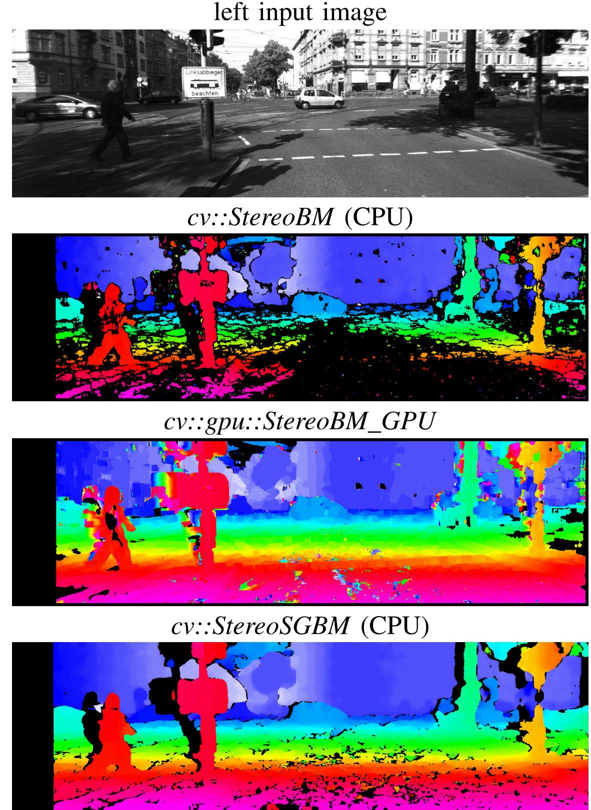

Figure 2. Stereo vision results of the *OpenCV* library: Purple encodes close, light blue distant objects.

Table I
STEREO VISION FRAME RATES OF THE *OpenCV* LIBRARY

<table>
<tr><th></th><th>Teebaum</th><th colspan="2">PC9F</th></tr>
<tr><td>cv::StereoBM (CPU)</td><td>47.76</td><td colspan="2">47.22</td></tr>
<tr><td>cv::gpu::StereoBM_GPU</td><td>146.12</td><td>91.93</td><td>27.38</td></tr>
<tr><td>gpu/stereo_multi.cpp</td><td>226.21</td><td colspan="2">42.43</td></tr>
<tr><td>cv::StereoSGBM (CPU)</td><td>3.45</td><td colspan="2">2.14</td></tr>
</table>

cv::gpu::StereoBM_GPU except for the filtering of intermediate results: *OpenCV's* GPU version hardly filters at all, while the CPU version discards most results on the ground because of their high ambiguity. The sample application *gpu/stereo_multi.cpp* utilizes two GPUs by statically partitioning the results to be computed into equal halves - this has proven to be beneficial to identical GPUs, but unfavorable for significantly different ones. Finally *cv::StereoSGBM* implements the slower yet popular *semi-global matching* [16].

B. Test Systems and Hardware Architectures

On the one hand this section introduces our specific test systems in order to allow interpretation and reproduction of the presented results. On the other hand it describes features and programming interfaces of the underlying hardware architectures, which are relevant to the following sections.

Table II shows the two heterogeneously equipped systems used for our experiments: *Teebaum* is a workstation with two identical performance-level CPUs and GPUs each. Despite its high power consumption under full load, similar systems

[1]please see section I.B for their specifications
[2]to be desribed in detail in section II

Table II
TEST SYSTEMS SPECIFICATIONS

processor model	core count and frequency	memory size and bandwidth
Teebaum		
Intel Xeon E5645 (2x)	2x 6 2.4 GHz	2x 6 GB 2x 32.00 GB/s
nVidia GeForce GTX 470 (2x)	2x 448 607 MHz	2x 1.25 GB 2x 133.9 GB/s
PC9F		
AMD Phenom II X6 1090T	6 3.2 GHz	8 GB 21.33 GB/s
nVidia GeForce GTX 460	336 675 MHz	1 GB 115.2 GB/s
nVidia GeForce GT 430	96 700 MHz	1 GB 28.8 GB/s

are successfully used in research prototype vehicles, e. g. autonomous cars [17]. An alternate view on common desktop PCs is provided by *PC9F*, which combines a single six-core CPU with both a performance- and an entry-level GPU.

Each of the CPUs is compatible with the *x86-64* architecture and several generations of SIMD instruction sets. The latter enable the simultaneous calculation up to 16 values per core, so a CPU's potential degree of parallelism is notably higher than its number of cores might suggest. Additionally, *Teebaum's* CPUs offer *Hyper-Threading*, i. e. a second set of registers per core to quickly switch contexts between idle and waiting threads. Other relevant properties of recent CPUs, like their comparably large and implicitly managed caches, have already been described in [4]. Regarding programming interfaces, we exploit data element-parallelism via *OpenMP* [19] and the *GNU C compiler's* SIMD intrinsics, which are compatible to *Intel's* [18]. The *Boost.Thread* library – wrapping *POSIX* threads and mutexes [20] on Linux – enables data flow- and task-parallelism.

At general-purpose GPU computing, the most common programming interfaces are *OpenCL* [21] and *C for CUDA* [22], with wrappers available for other languages [23]. *CUDA* is well-integrated with *OpenCV*, thus saving fundamental programming work for computer vision applications. The GPUs of our test systems contain between 96 and 448 scalar cores. These are however not completely independent: Groups of 32 are each hosted on one streaming multiprocessor (SM) and can only execute identical instructions simultaneously while branching needs to be serialized. Even though each core is scalar, we will show how to achieve parallelism via SIMD within a register (SWAR) [24]. In order to hide latencies, GPUs demand a degree of parallelism much higher than their number of cores [22]. Their limited amount of registers and on-chip caches per thread needs to be sparingly used and in some cases explicitly managed by programmers.

II. ALGORITHMS AND DATA ELEMENT-PARALLELISM

Practically all stereo vision implementations consist of a sequence of independent processing steps rather than a single function. We have already presented our approaches to exploit parallelism for either CPUs or GPUs in [4], and will therefore only summarize them to the extent required for understanding the heterogeneous methods in section III.

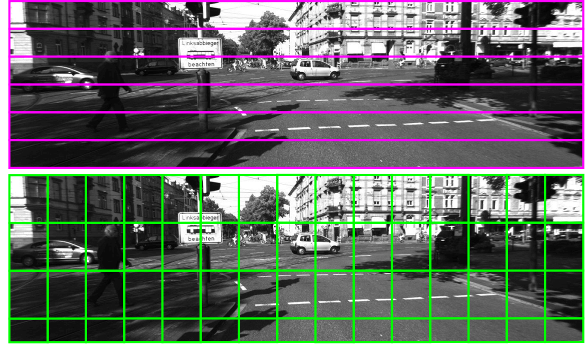

Figure 3. Device-specific partitioning scheme for CPU cores (purple) and SMs of GPUs (green)

There are two important common properties of the following processing steps: A given result pixel is generally independent from its neighbors, and only local regions of interest (ROI) of the input images need to be read in order to compute it. Nevertheless, efficient parallel algorithms share intermediate results, and their implementations account i. a. for cache locality. Figure 3 depicts the device-specific partitioning schemes derived from these conditions: The typical row-major memory layout of images is in favor of horizontal partitions, each of which is processed on an individual CPU core. In contrast, GPU partitions have a fixed size which is determined by the resources available on a single SM and required by each algorithm. Partitions are successively assigned to SMs until none are left.

A. Undistortion and Rectification

Removing lens distortion from camera images allows to apply the pinhole camera model [25] for mathematically simple re-projection of pixels back to their 3D coordinates. Additionally, rectification virtually adjusts focal lengths and aligns image planes, such that every spot of the scene appears at the same row within the left and right image – this drastically reduces the complexity of the later matching step. The corresponding parameters can be obtained by offline calibration [26] and used to compute static lookup tables (LUT). These are applied in conjunction with bi-linear interpolation to map camera pixels to rectified pixels:

$$I_{rect}(x, y) = I_{cam}\left(LUT_x(x, y), LUT_y(x, y)\right)$$

At this step, our code wraps *OpenCV* on both CPU and GPU: The code for the former is a well-optimized single-core implementation, which we extended towards multiple cores by applying the partitioning scheme mentioned above. Its GPU code is already parallel by design and utilizes the hardware texturing units for very fast interpolation. Hereby, this processing step's impact on running time becomes secondary and further optimization is not required.

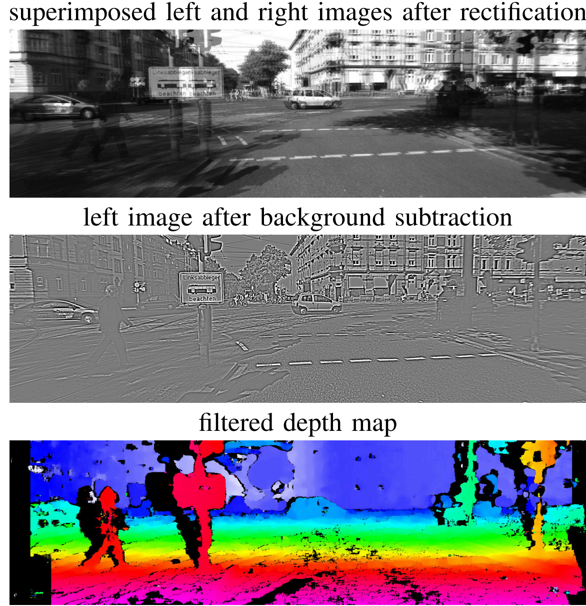

Figure 4. Intermediate and final results of our stereo vision method

B. Background Subtraction

A given object may appear differently bright on the left and right images due to variations in exposure, aperture and gain of each camera. This potential cause of errors can be mitigated by applying a high-pass filter, i. e. by blurring the image with a Gaussian kernel ($\sigma = 3.0\ px$) and subtracting it from the original one:

$$I_{BS}(x,y) = I_{rect}(x,y) - \sum_{j,k=\lfloor -3\sigma \rfloor}^{\lceil 3\sigma \rceil} I_{rect}(x+k, y+j)\frac{e^{-\frac{k^2+j^2}{2\sigma^2}}}{2\pi\sigma^2}$$

OpenCV makes use of the fact that this filter is separable, but does not apply any further optimizations or CPU parallelization. Consequently, running times are within the same order of magnitude as the actual stereo matching step. On both target architectures, we approximate the filter coefficients by integers and scale them such that their sum equals 256. Together with 8-bit images, this has several advantages:

- No floating point conversions are required.
- Normalization of the sum term is possible via bit-shifting instead of slow integer division.
- 16-bits are sufficient for intermediate sums, allowing each CPU or GPU core to compute 8 or 2 horizontally neighboring pixels via SIMD or SWAR respectively.

As an example of SWAR, the following C code performs a pair-wise addition and normalization as specified above[3]:

```
uint16_t pixel[2], sum[2];
*(uint32_t*)&sum += *(uint32_t*)&pixel;
*(uint32_t*)&sum = 0x00FF00FF &
     ( *(uint32_t*)&sum >> 8 );
```

[3]For the addition to be correct, no bit must be carried from the lower to the upper word, i. e. the 16-bit types must not overflow. The logical AND is required to set bits shifted from the upper into the lower word to zero.

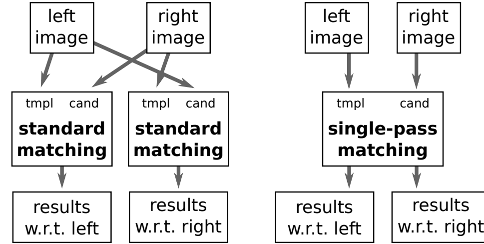

Figure 5. Modes to obtain matching results w. r. t. both left and right image

Since every such group of pixels can be computed independently, multi-core and whole-GPU parallelization is straightforward – GPU partitions are sized $64 \times 24\ px$ here.

C. Stereo Matching and Filtering

After rectification, the distance Z of a given object depends on the focal length f of the cameras, the baseline width B between them and the disparity Δx, i. e. the difference in pixel columns at which the object appears in each of the stereo images:

$$Z(x,y) = \frac{fB}{\Delta x(x,y)}$$

Our method of choice for densely estimating Δx is block matching, as it allows efficient parallel implementations with an easily cacheable memory footprint [4]. Descriptively, this method tries to recognize small template windows from one image within a set of candidates from the other image. Mathematically, the sum of absolute differences (SAD) between template and candidate window is minimized:

$$\Delta x_{tmpl}(x,y) = \arg\min_{\Delta x} SAD_{tmpl}(x,y,\Delta x)$$

$$SAD_{tmpl}(x,y,\Delta x) = \sum_{j,k \in window} |I_{tmpl}(x+k, y+j) - I_{cand}(x+k-\Delta x, y+j)|$$

Similar to a mean filter, the complexity of computing adjacent SADs can be made independent from window size via running sum tables [4], [13].

Alongside these advantages, the resulting disparities often contain some errors and therefore require filtering. Since both the left and right image can be used as template and the respective other as candidate, requiring both results to be consistent has proven most effective:

$$\Delta x_{left}(x,y) \stackrel{!}{=} -\Delta x_{right}\left(x - \Delta x_{left}(x,y), y\right)$$

As Figure 5 indicates, this does not neccessarily require two standard matching steps with switched input image roles: Alternatively, optimal Δx_{left} and Δx_{right} may be found in a single pass by interpreting each SAD w. r. t. both left and right image:

$$SAD_{left}(x,y,\Delta x) = SAD_{right}(x - \Delta x, y, \Delta x)$$

Both our CPU and GPU implementations apply the above algorithmic simplifications and the initially-mentioned partitioning scheme across multiple cores and SMs [4]. However, improvements on this previous work include the following:

- 8x SIMD on CPUs and 2x SWAR on GPUs are used to evaluate multiple values of Δx simultaneously.
- We deviate from the initial scheme by partitioning both horizontally and vertically: Even though cache locality is in favor of horizontal partitions, our implementation saves cache size for narrower vertical ones. Section IV.A will show that this is important to scaling across more than a few CPU cores.

III. Strategies for Heterogeneous Systems

As a foundation for this sections, we will review [27]'s categorization of parallelism and relate it to the presented task of stereo vision via Figure 1:

- Indirectly, data element-parallelism has already been introduced above. Given a single processing step, it generally refers to the decomposition of its result into items to be computed simultaneously. For most of our stereo vision algorithms, this pattern can be applied even to single pixels, even though Figure 1 only indicates independent image columns for simplicity.
- Data flow-parallelism may occur if input or intermediate data passes through a series of processing steps: Transferred to our application, the nth image pair may already be rectified on one device while others perform background subtraction on pair $n-1$ and stereo matching on pair $n-2$ respectively.
- Task parallelism in general denotes the availability of separate jobs to be computed independently. A stereo vision job consists of applying all processing steps to a single pair of images. Therefore, task- and data flow-parallelism are similar with regard to multiple work items being processed at a time, but differ by exclusively assigning every device to either a processing step or a work item.

For each of these categories, the following sub-sections describe a method to apply it to multiple heterogeneous processors, including algorithmic requirements and adaptation strategies. Figure 6 visualizes the way each strategy maps to a different category.

A. Partitioning

Partitioning extends data element-parallelism across different devices and is therefore most comparable to the previously-mentioned *split frame rendering*: Only one item is computed at a time, but work is distibuted among k participants. This allows dependencies between consecutive work items, but also requires splitting and re-combining an item to be efficiently possible. By cutting images horizontally, the latter is simple for our application.

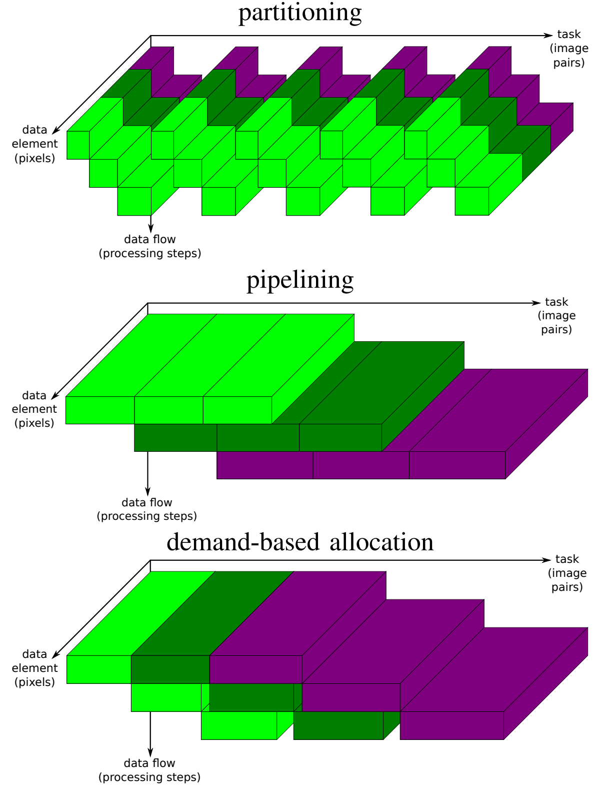

Figure 6. Mapping of parallelization categories to heterogeneous devices: Each color indicates an individual processor.

Assuming that throughput is maximized by avoiding idle processors, an optimal partitioning scheme allows all processors to finish their partitions at the same time. To automatically achieve this state, our implementation initially assigns an equally-sized fraction $r_j = 1/k$ of the 1st work item to each device $j \in [1, k]$. The corresponding running times t_j are regularly measured and used to calculate $i_j = r_j / t_j$, i. e. the extrapolated throughput of full work items per device. This noisy measure is finally tracked by a discrete PT_1 smoothing filter with damping $\alpha \in [0, 1]$ and used to calculate the ratios for partitioning the next work item:

$$\hat{i}_j = \alpha \hat{i}_j + (1-\alpha) i_j \qquad r_i = \frac{\hat{i}_j}{\sum_{j=1}^{k} \hat{i}_j}$$

Obviously these calculations need to be done by a central instance which schedules tasks in a push-based fashion.

B. Pipelining

Pipelining is a standard method to utilize data flow-parallelism. We have already investigated it for homogeneous multi-core systems in [4] and found it to be advantageous on a 48-core *x86*-server and a 64-core embedded platform. Algorithmically, pipelining only requires at least one processing step per device. Concerning implementations however, if k participating processors do not share their memory, every full work item needs to be copied at least $k-1$ times. Furthermore, the assignment of processing steps

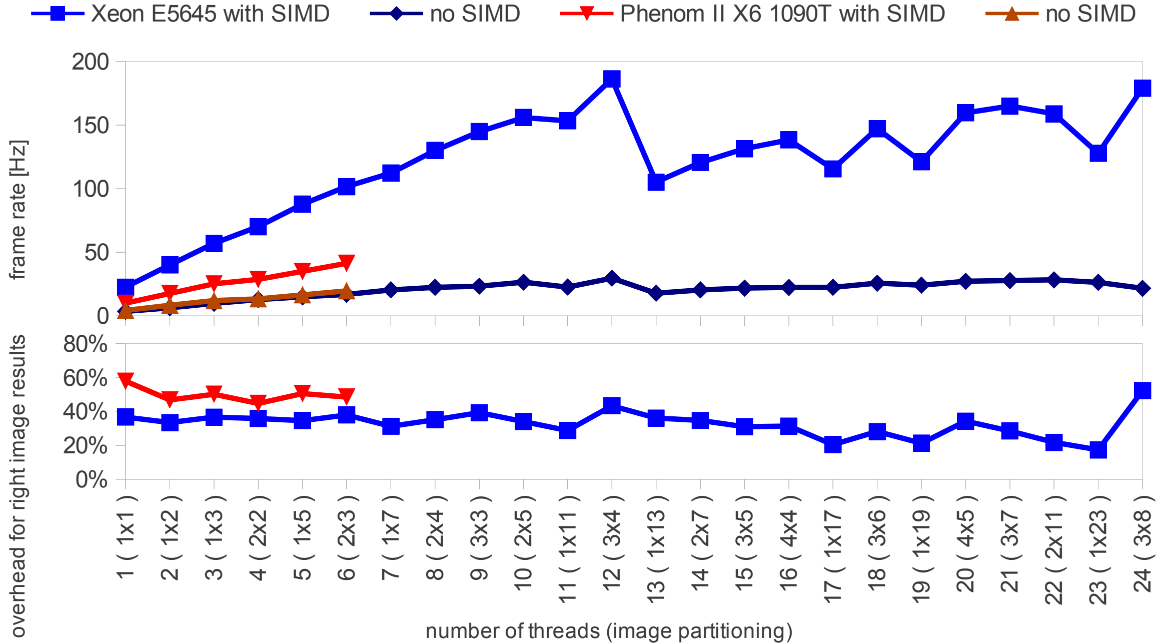

Figure 7. Scaling of the CPU stereo matching step (top); overhead of additionally computing results w. r. t. the right image (bottom)

to devices potentially is a complex optimization problem for which we cannot yet offer a complete and automated solution. With our application having three processing steps and our experiments using two or three devices however, there are only a total of six possible pipelining configurations. From these, we heuristically select the one minimizing the largest idle time among all devices.

C. Demand-based Allocation

By finally exploiting task-parallelism, demand-based allocation is a generalized form of *alternate frame rendering* with a major difference: Instead of actively pushing work items to devices via round-robin or any other fixed or pre-computed pattern, it implements a queue which passively waits for any device to complete its current job and pull a new one. Because jobs should ideally be processed independently, dependencies between consecutive work items are not impossible but still disadvantageous. An advantage is that computational load is automatically balanced between differently performing devices without any central instance.

IV. Experimental Evaluation

This section will evaluate the previously proposed methods in two stages: Initially we present the performance of our implementations in an architecture-specific way and investigate the effects of the optimizations discussed in section II. On this basis, we apply the strategies for utilizing and adapting to heterogeneous processing units introduced in section III. All experiments have been conducted with [12]'s publicly available data set sized $1344 \times 391\ px$ and $\Delta x \in [0, 112)$.

Table III
Effect of vertical and horizontal partitioning on CPU matching step frame rates

Teebaum		PC9F	
partitions	frame rate [Hz]	partitions	frame rate [Hz]
1x24	128.69	1x6	39.31
2x12	169.33	**2x3**	**41.31**
3x8	**179.03**	3x2	38.47
4x6	177.53	6x1	32.06
6x4	169.41		
8x3	113.73		
12x2	84.87		
24x1	59.39		

A. Architecture-specific Optimization

We will focus on the single processing step of stereo matching here for two reasons: It accounts for the majority of our application's running time, and most of the insights found at it may be transferred to the simpler previous and subsequent steps. Figure 7 visualizes several aspects of our CPU implementation[4]:

- Both test systems scale almost linearly up to their actual number of cores. Table III however shows that a good compromise between cache locality (i. e. horizontal partitions) and required cache size (vertical partitions) needs to be found in order to achive this scaling.
- SIMD is very effective on the *Xeon* platform, while the *Phenom II* must emulate two instructions [4], [28].
- Additionally placing threads on *Teebaum's* 13th to 24th virtual *HyperThreading* cores has a strong impact on

[4]With previous and subsequent steps not included, frame rates should not be interpreted as those of a final application. Compard to running times however, they give an improved impression on scaling across multiple cores.

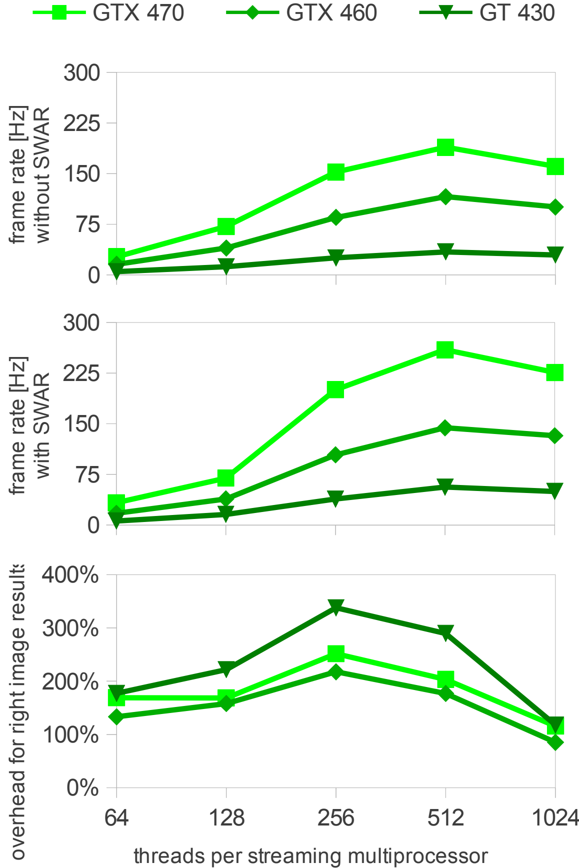

Figure 8. Effect of the number of threads without (top) and with 2x SWAR (middle) of the GPU stereo matching step; overhead of additionally computing results w. r. t. the right image (bottom)

performance, so obviously their initialization overhead outweighs the compensation of occasional idle states.

- Finding the disparity map Δx_{right} in addition to Δx_{left} within a single matching pass increases running times by only 20 to 60 %, which is clearly preferable to a separate second pass with switched image roles.

Since single SMs of a GPU cannot be disabled, scaling is not directly observable here. Nevertheless, Figure 8 allows several interesting observations to be made:

- A SM always computes $64 \times 48 = 3072\ px$ of results using a specified number of threads. Each of them initializes one SAD from scratch and then efficiently derives several neighbors via running sum tables. Fewer threads therefore cause less initialization overhead, while more of them help hide latencies. A clear optimum of 512 can be found for all GPUs tested.
- 2x SWAR increases performance by about one third.
- While iterating over the possible range of Δx, its so far best values w. r. t. the template image can be kept in thread-private registers. In contrast, those w. r. t. the candidate image need to be accessed by concurrent threads via atomic operations. Expectedly, Δx_{left} and Δx_{right} should be determined by separate GPU passes.

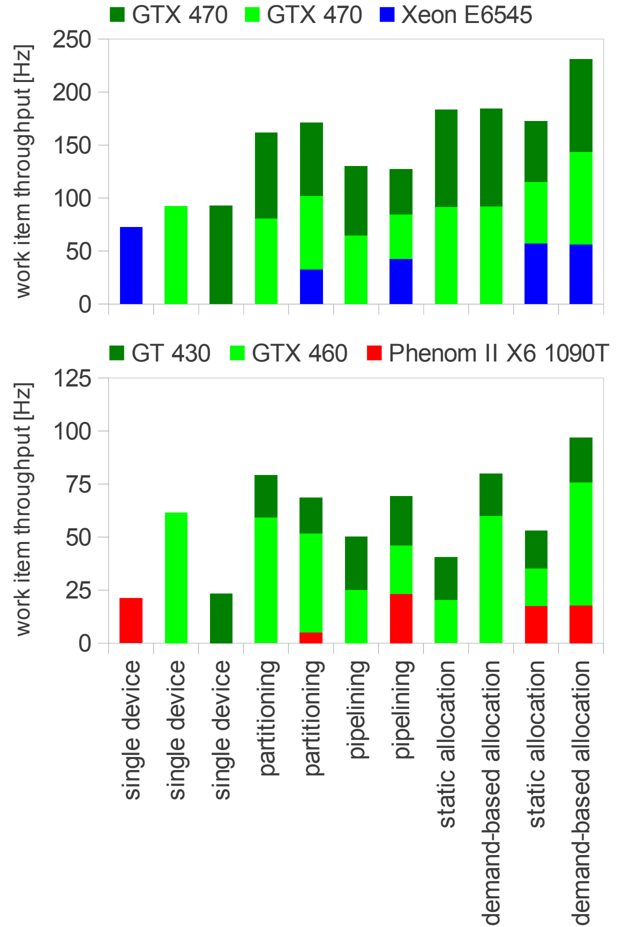

Figure 9. Stereo vision frame rates of different heterogeneous parallelization strategies for *Teebaum* (top) and *PC9F* (bottom): Stacked bars indicate the contribution of each processor to the overall performance.

B. Run-time Adaptation to Heterogeneous Processing Units

It is finally time to go beyond specific processors or single processing steps and evaluate the presented strategies for utilizing and adapting to heterogeneous processing units. Common boundary conditions of our experiments include the use of SIMD and SWAR in every possible processing step and the interpretation of the available CPU cores as a single device. Stereo matching results w. r. t. left and right image are obtained from one CPU but two GPU passes.

For each of our test systems, Figure 9 shows frame rates of various configurations: Supplementarily, that of each single device is given as a baseline. The previously-introduced strategies partitioning, pipelining and demand-based allocation are applied twice each – using only GPUs as well as using both GPUs and CPUs. Static allocation implements a round-robin scheduling for comparison with our demand-based method. Several conclusions can be drawn:

- Partitioning maintains a reasonable scaling especially for GPUs, which proves the overhead of splitting and re-combining images to be low. CPUs however tend to become less efficient when working on small partitions.

- Pipelining throughput is notably limited by its slowest stage, which regularly forces other processors to idle. The additional latencies due to frequent memory transfers can be mitigated to some extent by overlapping them with computations [22].
- Demand-based allocation especially benefits differently-performing devices, but also causes no measurable overhead when being applied to identical processors.

Conclusively, all of the proposed adaptive strategies are able to improve throughput. Latency however can by design be decreased solely by partitioning, at which all processing power is focused on a single work item. This is only a minor advantage over pipelining and demand-based allocation, considering the frame rates in connection with the fact that at most three image pairs are processed at a time.

In order to gain further insights on adaptation, Figure 10 plots the automatically adjusted partitioning ratios r_j of all devices for 800 consecutive work items. A steady state is already reached after about 20 items, but may be disturbed by short phases of competing CPU utilization e. g. by the operating system. Halfway throughout the sequence we purposefully applied additional load on the primary GPU by starting the *nbody* example from the *CUDA* SDK: On *Teebaum*, its work is completely taken over by the other similarly-performing devices, while *PC9F's* fastest processor still keeps contributing to stereo vision significantly.

V. Conclusions and Outlook

We have presented parallel implementations of dense matching-based stereo vision for both multi-core CPUs and GPUs. Relevant optimizations w. r. t. both algorithms and implementations have been explained, verfied and interpreted. Our implementations are able to meet real-time constraints and to compete with *OpenCV*, a widely-used and well-optimized computer vision library. Furthermore, we have introduced and experimentally evaluated three strategies for cooperatively utilizing heterogeneous processors for general-purpose computations. Especially partitioning and demand-based allocation have proven to effectively adapt to our test systems at run-time. Because the strategies require partially oppossing properties from algorithms, at least one or two are likely to be suitable for many parallel computing tasks.

Future works will focus on two different aspects: Algorithmically, we are investigating alternatives to the proposed filtering method which do not demand both left and right matching results – especially GPU performance will benefit if the second matching pass can be avoided. Concerning the source code associated with this paper, the proposed parallelization strategies added a noteworthy amount of complexity. Since these strategies are transferable to many other tasks offering independent jobs or partitionable work items, it seems reasonable to encapsulate them in a separate library. Here both an object-oriented syntax and a domain-specific language could simplify the source code of applications.

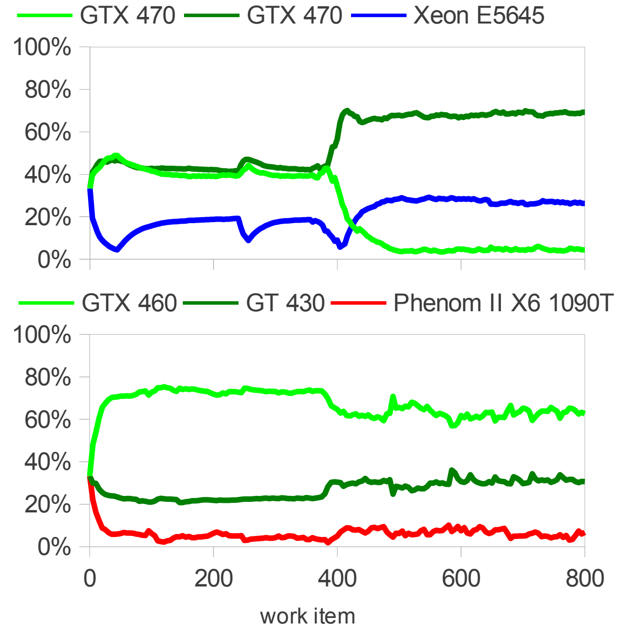

Figure 10. auto-adaptation of partitioning ratios to a competing GPU application for systems A (top) and B (bottom)

References

[1] Continental AG, *Two Eyes Are Better Than One – The Stereo Camera*. www.conti-online.com/generator/www/com/en/continental/pressportal/themes/press_releases/3_automotive_group/chassis_safety/press_releases/pr_20110504_stereo_camera_en,version=2.html, May 2011.

[2] M. Elmezain, A. Al-Hamadi, and B. Michaelis, *Real-Time Capable System for Hand Gesture Recognition Using Hidden Markov Models in Stereo Color Image Sequences*. International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision Proceedings, February 2008.

[3] K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley* University of California at Berkeley Technical Report No. UCB/EECS-2006-183, December 2006.

[4] B. Ranft, T. Schönwald, and B. Kitt, *Parallel Matching-based Estimation – a Case Study on Three Different Hardware Architectures*. IEEE Intelligent Vehicles Symposium, May 2011.

[5] D. Delling et al., *PHAST: Hardware-Accelerated Shortest Path Trees*. Microsoft Research Technical Report MSR-TR-2010-125, September 2010.

[6] R. Bordawekar et al., *Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application!*. IBM Reseach Report RC25033, April 2010.

[7] NVIDIA Corporation, *SLI Best Practices*. developer.nvidia.com/sli-best-practices, February 2011.

[8] J. Singh, and I. Aruni, *Accelerating Smith-Waterman on Heterogeneous CPU-GPU Systems*. International Conference on Bioinformatics and Biomedical Engineering Proc., May 2011.

[9] A. Nere, A. Hashmi, and M. Lipasti, *Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms*. IEEE International Parallel and Distributed Processing Symposium Proceedings, May 2011.

[10] R. Ferrer et al., *Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL*. International Workshop on Languages and Compilers for Parallel Computing Proceedings, October 2010.

[11] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, *Ocelot: A Dynamic Compiler for Bulk-Synchroneous Applications in Heterogeneous Systems*. International Conference on Parallel Architectures and Compilation Techniques Proceedings, September 2010.

[12] A. Geiger, M. Roser, and R. Urtasun, *Efficient Large-Scale Stereo Matching*. Asian Conference on Computer Vision Proceedings, November 2010.

[13] P. Azad, T. Gockel, and R. Dillmann, *Computer Vision: Principles and Practice*. Elektor Electronics, 2008.

[14] D. Gallup, J. Frahm, and J. Stam, *CUDA Stereo*. www.cs.unc.edu/˜gallup/cuda-stereo, February 2012.

[15] G. Bradski, *The OpenCV Library*. Dr. Dobb's Journal of Software Tools, 2000.

[16] H. Hirschmüller, *Stereo Processing by Semi-Global Matching and Mutual Information*. IEEE Transactions on Pattern Analysis and Machine Intelligence, volume 30(2), February 2008.

[17] A. Geiger, M. Lauer, F. Moosmann, B. Ranft, H. Rapp, C. Stiller, and J. Ziegler, *Team AnnieWAY's entry to the Grand Cooperative Driving Challenge 2011*. IEEE Transactions on Intelligent Transportation Systems, to be published.

[18] Intel Corporation, *Intel C++ Intrinsics Reference*. software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/index.htm#intref_cls/common/intref_overview.htm, December 2011.

[19] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 3.1*. www.openmp.org/wp/openmp-specifications, July 2011.

[20] B. Nichols et al., *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly, 1996.

[21] Khronos OpenCL Working Group, *The OpenCL Specification*. www.khronos.org/registry/cl, November 2011.

[22] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*. developer.nvidia.com/cuda-toolkit-41, December 2011.

[23] A. Klöckner, *PyCUDA: Even Simpler GPU Programming with Python*. GPU Technology Conference Proceedings, September 2010.

[24] R. J. Fisher, *General-Purpose SIMD within a Register: Parallel Processing on Consumer Microprocessors*. PhD thesis, Purdue University, January 2003.

[25] R. Hartley and A. Zisserman, *Multiple View Geometry in computer vision, second edition*. Cambridge University Press, 2008.

[26] J.-Y. Bouguet, *Camera Calibration Toolbox for Matlab*. www.vision.caltech.edu/bouguetj/calib_doc/index.html, July 2010.

[27] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley, 2004.

[28] Advanced Micro Devices Inc., *SSEPlus Project Overview*. sseplus.sourceforge.net/SSEPlus.pdf, May 2008.