A Stream Processing Framework for On-line Optimization of Performance and Energy Efficiency on Heterogeneous Systems

Benjamin Ranft, Oliver Denninger
FZI Research Center for Information Technology
76131 Karlsruhe, Germany
Email: {ranft,denninger}@fzi.de

Philip Pfaffe
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
Email: philip.pfaffe@kit.edu

Abstract—Modern processors have the potential of executing compute-intensive programs quickly and efficiently, but require applications to be adapted to their ever increasing parallelism. Here, heterogeneous systems add complexity by combining processing units with different characteristics. Scheduling should thus consider the performance of each processor as well as competing workloads and varying inputs.

To assist programmers in facing this challenge we present libHawaii, an open source library for utilizing heterogeneous systems easily and efficiently. It supports exploiting data flow, data element and task parallelism via pipelining, partitioning and demand-based allocation of consecutive work items. Scheduling is automatically adapted on-line to continuously optimize performance and energy efficiency. Our C++ library does not depend on specific hardware architectures or parallel computing frameworks. However, it facilitates maximizing the throughput of compatible GPUs by overlapping computations and memory transfers while maintaining low latencies.

This paper describes the algorithms and implementation of libHawaii and demonstrates its usage on existing applications. We experimentally evaluate our library using two examples: General matrix multiplication (GEMM) is a simple yet important building block of many high-performance computing applications. Complementarily, the detection, extraction and matching of sparse features within images exhibits inter alia nondeterministic memory access and synchronization.

Keywords—heterogeneous computing; stream processing; load balancing; energy efficiency; real-time; parallel programming;

I. INTRODUCTION

Throughout the last years, standard processors have been enhanced particularly by incorporating an increasing number of parallel processing units. At the same time, the previously common growth of serial performance is inhibited by power consumption, memory latencies and limited instruction-level parallelism. Thus, not only developers of high-performance or real-time software need to create and optimize parallel algorithms and implementations which scale well with the number of processing units available.

Additionally there is a trend of systems becoming more and more heterogeneous by including multiple processors with different characteristics: The best-known example are general purpose graphics processing units (GPUs), which perform well at problems with sufficient data parallelism. Their advantage over CPUs regarding throughput and energy efficiency has been found to be significant in several domains [1][2][3]. This holds even when utilizing all CPU cores along with their SIMD capabilities and when also considering data transfers to and from discrete GPU memory. Nevertheless, reports of hundredfold speed-ups often neglect these aspects.

Data locality is less crucial if CPU and GPU are integrated and share a unified memory [4][5], but it remains important to the discrete accelerators prevalent in high-performance computing. Aside from this typical combination, heterogeneity also appears in compatible cores of a single processor which are designed for either performance or efficiency [6].

The aforementioned trend towards heterogeneous systems poses requirements to software development, based on which we defined the goals below for libHawaii1. The methods and approaches used to achieve them constitute the contributions of this paper. We specifically target streaming applications, which apply a set of filters to successive work items – a common characteristic of scientific or real-time applications.

- Generality: To enable the use of all available processors by as many applications as possible, all their contained task, data and data flow parallelisms [7] are exploitable.
- Performance portability: While automatically adapting parallelism throughout runtime, libHawaii incorporates a system’s hardware, the application itself as well as competing processes. At this, we evaluated the feasibility and potential of different approaches tightly coupled within a single program [8] before creating libHawaii.
- Energy efficiency: Real-time applications depend on a sufficient rather than the highest possible throughput. In this case, we optimize power instead of performance by preferring more efficient processors while still meeting the throughput requirement. The power consumption of each pair of processor and filter is estimated from a model due to the lack of appropriate sensors in ordinary systems.
- Productivity: Our library relieves programmers from manually tuning applications for various combinations of processors, parameters and inputs. Since it is agnostic on the internals of a user’s code and does not

1This acronym stands for “heterogeneous adaptive work allocation implementation items”. It has first been applied to our internally-used computer vision library libToast2, the “tools for the analysis of stereo images”.
employ any custom compiler or runtime environment, interfacing usually does not require any modifications but only few additional lines of code. Being an open source² C++ library, libHawaii uses the same language as many performance-critical applications and may be extended towards any special requirement if necessary.

This paper’s remainder is organized as follows: Section II presents related work with a focus on existing parallel computing frameworks and differentiates between these and our contribution. Section III describes libHawaii’s interface and implementation, especially its strategies for continuously optimizing performance and energy efficiency. Section IV employs sample applications to evaluate our library w.r.t. the goals above and illustrates the procedure of interfacing it with existing applications via code examples. Section V concludes the paper and presents an outlook on future works.

II. RELATED WORK AND CONTRIBUTION

The next paragraphs present increasingly versatile parallel computing approaches: Mainly processor vendors offer frameworks for offloading computations to either multi-core CPUs or accelerators. Improvements include code portability and cooperation of heterogeneous processors. Still, generic scheduling optimization remains an active research topic.

The introduction of GPUs to general purpose computing was accompanied by architecture-specific libraries containing commonly used functions. They simplify switching from CPUs by implementing interfaces familiar from e.g. the C++ Standard Library [9] or Intel IPP [10]. Libraries like OpenCV [11] allow choosing between similar functionality for CPUs and GPUs. Data parallelism within own code can be conveniently exploited by multi-core CPUs and GPU-like accelerators through the compiler directives of OpenMP, OpenACC [12] or OmpSs [13]. [14] facilitates handling multiple CUDA-capable GPUs and irregular workloads.

Today’s best-known framework for heterogeneous computing is OpenCL: A kernel implemented in its extensions of the C language is portable to each compatible processor. [15] extends the scope to clusters by mapping remote nodes to virtual local devices. In spite of the above portability, achieving best performance still requires processor-specific tuning. [16] and [17] automatically generate optimized GPU code from StreamIt and Lime language sources respectively. [18] also supports CPUs, but is limited to stencil computations. In conclusion, the above and other frameworks can be used to achieve per-processor parallelism before utilizing all processors of heterogeneous systems via libHawaii.

Several publications have complemented the above portability with concurrent execution on heterogeneous processors: [19] and [20] find performance models for two different processors from an initial training run in order to optimally split data and computations, which results in latencies close to a manually-optimized mapping. Independence from training runs and hence the possibility of on-line adaptation is demonstrated by [21] for OpenMP loops. Our partitioning strategy is similar, but employs a more complex performance model and allows generic expressions of data parallelism. Instead of splitting data and computations into one ideally-sized partition per processor, [22] and [23] allocate many smaller chunks: This may add overhead, but achieves adaptation implicitly as faster processors consume chunks.

If an algorithm can be expressed by simple patterns like MapReduce, heterogeneous data parallelism can be realized using skeletons: [24] or [25] implement such patterns and offer automatic memory management. [26] combines skeletons or high-level functions into a macro data flow graph which is processed to heterogeneous. Neither the aforementioned frameworks nor our library match the skeletons’ conciseness, but are more flexible w.r.t. suitable algorithms.

Optimizing energy efficiency usually requires either measuring power consumption with additional sensors [27] or estimating it e.g. from artificial neural networks based on hardware performance counters [28]. Mappings from each processor’s voltage/frequency state to its power consumption are employed by [29] to maximize a heterogeneous system’s throughput at a given power budget. [30] finds a compromise between performance and power by optimizing the energy-delay product metric. In contrast, [31]’s and our goal is to achieve a given throughput as efficiently as possible. Our power model’s unique feature is providing estimates not only per processor, but for each pair of processor and filter.

When shifting the focus from stream processing to data flow and task parallelism, Intel TBB [32] is commonly used: It can dynamically map a dependency graph of tasks to CPU cores, but does not support accelerators equally well. Other frameworks specifically target heterogeneous systems, e.g. [33] which nevertheless requires a shared address space. [34] makes scheduling decisions based on the relative speed-up between CPU and GPU. A comparative overview of scheduling heuristics for independent tasks is given by [35]. [36] introduced the popular heterogeneous earliest-finish-time algorithm. It is used among others by the PEPPER component model [37] or the StarPU runtime system [38], which optionally take into account pre-defined or history-based performance models. The extensive XPU/MHPM framework [39] supports the same three kinds of parallelism as our library through an interface which also requires only few application-specific lines of code. libHawaii however includes two additional noteworthy features: The assignment of tasks to processors is automatically adapted on-line, and not only performance but also energy efficiency can be optimized. This second aspect has mainly been studied beyond our scope of single applications: [40] distributes recurring tasks among two heterogeneous processors, which is relevant e.g. for real-time operating systems. On a similar
level, [41] and [42] manage all enqueued tasks such that the power states of processors or whole nodes can be reduced without affecting throughput.

III. IMPLEMENTATION AND SCHEDULING STRATEGIES

We now present the algorithms, implementation and usage of \textit{libHawaii} in detail. Section III-A covers its fundamental building blocks and their interfaces to a user’s application. Our library’s classes for combining these blocks to adaptively optimize performance and efficiency on heterogeneous systems are described in section III-B. Finally, section III-C introduces supplemental functionality for easily and effectively optimizing performance and energy efficiency on a heterogeneous system. A user’s application-specific processing steps can now be combined using different methods to achieve optimized performance and energy efficiency on a heterogeneous system, a single filter instance’s energy per work item can be estimated:

\[ w = \sum_i P_i \Delta t_i \]  

3) Base Flow Component: \textit{libHawaii}’s interface to application code is now complete, but one fundamental building block remains: As fig.1 indicates, \textit{Flow}.::\textit{pull()} and \textit{push()} implement the interface for acquiring and passing work items. While doing so, this abstract class can measure performance metrics such as latency and throughput \( f \) for evaluation by derived classes. This information may be generated and accessed concurrently, so lock-free implementations minimize runtime overheads. We apply exponential filtering to obtain smooth estimates \( \hat{f} \) based on noisy measurements \( f \) acquired from the \( k \)-th work item:

\[ f_k = \alpha f_{k-1} + (1 - \alpha) \hat{f}_k, \quad \alpha \in [0, 1) \]  

Apart from these measurements, an instance of \textit{Flow} can be configured to actively limit throughput. For evaluating an on-line application with recorded off-line data, this allows making inputs available at the original sensor’s rate. However, the key purpose of this feature is enabling the following strategies to optimize energy efficiency by throttling less efficient processors. Please note that these strategies also require an estimate of each filter’s potential maximum throughput: \textit{Flow} provides this by compensating not only for throttling, but also for empty input or full output queues.

B. Adaptive Heterogeneous Computing

A user’s application-specific processing steps can now be combined using different methods to achieve optimized performance and energy efficiency on a heterogeneous system. We have laid the foundation for this part of \textit{libHawaii} in [8] by evaluating the feasibility and potential of the strategies below tightly coupled within a single prototype application. Therefore, this section repeats only core statements and otherwise focuses on added improvements such as energy efficiency, generalized implementations, and extensions.

\footnote{Using Linux’s \texttt{getusage()} call for CPUs and the event API of GPUs.}
Fig. 2 shows that our library offers one heterogeneous computing strategy corresponding to each orthogonal kind of parallelism constituted e.g. in [7]: Demand-based allocation exploits task parallelism while pipelining and partitioning make use of data flow and data parallelism respectively. The UML diagram in fig. 1 reveals that the classes Pipeline, DemandAlloc and Partitioning not only inherit from Flow, but also can wrap instances of this base type. This allows users to not only combine processing steps but also to nest different strategies. Example configurations will be illustrated in section IV after explaining each strategy:

1) Pipelining: The Pipeline class implements a standard approach: If an application processes a stream of work items in multiple steps, each of them is mapped to one specific processor. The set of possible mappings is therefore not only countable, but often limited to very few combinations – this interferes with precise adaptation to the conditions present in a heterogeneous system. For that reason and due to the merely moderate results of pipelining in [2] and [8], Pipeline is the only strategy not to implement automatic performance and energy optimization in the class itself. It is nevertheless very useful in conjunction with I/O-related processing steps such as reading from disk, receiving sensor data, displaying results or connecting to a middleware [43].

With those being typical starts and ends of processing, a Pipeline is usually used as a basis within which other heterogeneous parallelization strategies are nested. Queues are used to connect consecutive stages, i.e. filters or nested strategies. Each queue must be specified by the user for two reasons: It can and should have a limited capacity so that a high-throughput upstream stage is eventually throttled in case a downstream stage cannot keep pace with it. Also, we provide three different types: FIFO and priority queues as well as an ordered queue for use after a DemandAlloc.

2) Demand-based allocation: The class DemandAlloc regards work items as individual tasks which can be processed concurrently by different heterogeneous processors. It is therefore preferable to minimize synchronization between items, yet they are not required to be completely independent. Our implementation is able to optimize performance in an implicit way: The threads controlling each participating processor pull work items from a shared input queue and thereby effectively combine their throughputs. Because an earlier item assigned to a slow processor might be “overtaken” by a later item processed on a sufficiently faster one, the output queue needs to buffer items and only give them away ordered by their running number. Optimization of energy efficiency requires a more complex scheduling: For the contribution of each processor i to an instance of DemandAlloc, we regularly evaluate the energy it consumes per work item according to section III-A2. This metric is used to sort the processors by descending energy efficiency. Additionally, the minimum combined throughput \( f_{\text{min}} \) required to keep pace with the stream of input items is measured. Based on the processors’ unconstrained throughputs \( f_{\text{unc},i} \) the j most efficient ones can be spared from any throughput limit \( f_{\text{lim}} \):

\[
 f_{\text{lim},j} = \infty \forall j : \sum_{i=1}^{j} f_{\text{unc},i} < f_{\text{min}} \quad (3)
\]

The next best processor in terms of efficiency may only contribute as much as necessary and is therefore limited to:

\[
 f_{\text{lim},j+1} = f_{\text{min}} - \sum_{i=1}^{j} f_{\text{unc},i} \quad (4)
\]

To exclusively optimize energy efficiency, the remaining processors should actually not be used at all. However, our implementation only limits them to 5% of \( f_{\text{min}} \) in order to always gather updated performance measures from them.

3) Partitioning: Data parallelism can be expanded across multiple processors using Partitioning. Because all of them cooperate in processing a single work item at a time, this strategy can achieve the lowest latencies and is thus most suitable for real-time applications. Adaptation finds the ideal ratios for splitting an item into per-processor partitions. We have implemented two major extensions since [8]: Aside from optimizing energy efficiency, the underlying performance model for each individual processor has been refined. Its throughput \( f \) is not only assumed inversely proportional by a factor \( m \) to a partition’s fraction or ratio \( r \) of a full work item, but now also includes a constant per-item overhead \( c \):

\[
 1/f = mr + c \quad (5)
\]

Although this change may seem minor, it especially stabilizes the small \( r \) of slower processors. Jointly estimating both model parameters based on throughput as a single measure also requires a more complex algorithm: As a novelty in this domain, our implementation uses an individual Kalman filter [44] for each processor, which incorporates the partitioning ratio of the current work item \( k \) in its measurement matrix \( H_k \). Its state vector \( \hat{x}_k \) concatenates the most recently estimated model parameters. With the period 1/\( f_k \) being the time difference between finishing the last two work item partitions, the above performance model can be written as:
\[1/f_k = H_k \hat{x}_k = (r_k \ 1) (m_k \ c_k)^T \]  

A parameter update requires several helper variables: The internal constants \(Q\) and \(R\) define the noise (co-)variance of the state vector and the period respectively. The state error co-variance \(P_k\) measures the uncertainty of the most recently estimated model parameters. Finally, the Kalman gain \(K_k\) is an intermediate variable to facilitate updating the state estimate and its uncertainty based on the most recently completed work item partition’s \(r_k\) and \(1/f_k\):

\[K_k = P_{k-1}H_k^T (H_kP_{k-1}H_k^T + R)^{-1} \]

\[\hat{x}_k = \hat{x}_{k-1} + K_k (1/f_k - H_k \hat{x}_{k-1}) \]

\[P_k = (I - K_kH_k)P_{k-1} + Q \]

This algorithm is implemented within the Adapter class, via which Partitioning connects to each participating processor. To determine the partition ratios for the next work item, only the most recently estimated performance model parameters are used – we will therefore omit the item index \(k\) used above, but re-introduce the processor index \(i\). Like with demand-based allocation, processors are initially sorted by their efficiency, i.e. their energy consumption per full work item. Here we assume that energy is proportional to partition ratio. This strategy then attempts to keep pace with the stream of input items \(f_{\text{min}}\) in an energy-efficient way by first finding the maximum allowed work item ratio of each processor:

\[r_i = (1/f_{\text{min}} - c_i) / m_i \]

If \(\sum r_i \geq 1\) holds true, the ratios of the least efficient processors can be reduced until this sum equals one full work item. Otherwise, Partitioning may be a bottleneck for the incoming stream of work items and – to minimize this effect – falls back to optimizing performance: This is achieved by preventing any processor from idling, which is equivalent to equalizing their respective throughputs. Algorithmically, we create a system of linear equations to be solved for the partitioning ratios \(r_i\) via LU decomposition:

\[
\begin{pmatrix}
m_1 & -m_2 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & m_{n-1} & -m_n \\
1 & 1 & \cdots & 1 & 1
\end{pmatrix}
\begin{pmatrix}
r_1 \\
\vdots \\
r_{n-1} \\
r_n
\end{pmatrix}
= \begin{pmatrix}
c_2 - c_1 \\
\vdots \\
c_n - c_{n-1}
\end{pmatrix}
\]

For a total of \(n\) processors, the first \(n - 1\) rows ensure pairwise equal throughputs, while the last row states that exactly one full work item is to be partitioned.

An equilibrium of ratios emerges iteratively as the adaptation algorithm is applied to consecutive work items. It is in this respect similar to the Newton-Raphson method for finding a function’s roots. This also explains why the performance model in equation (5) does not necessarily have to be valid globally for any processing step, but only locally at its current partition ratio or “operating point”. Partitioning is therefore well-suited even for filters whose computational complexity does not depend just linearly on input size.

\[C.\ Supplemental\ Functionality\]

As stated initially, our library does not require the use of a specific software framework or architecture. Nevertheless, it includes optional functionality making it particularly efficient to use discrete-memory GPUs compatible with the CUDA framework. Since they offer a higher bandwidth than memory shared with any CPU, such GPUs are prevalent in heterogeneous high-performance systems. The price of this advantage is that data usually needs to be copied to and from this discrete memory – the associated impacts on both runtime and code complexity can fortunately be mitigated:

1) GPU Processing Step: GPUs can perform computations and memory transfers concurrently, which enables exploiting data flow parallelism. While a generic Pipeline could be used for this purpose, the specific filter template StepBaseGPU allows doing so more concisely and deterministically: It requires the upload, compute and download parts of processing to be implemented separately, so they can be scheduled to overlap for consecutive work items. Fig. 3 shows that the sum of upload, compute and download time remains a lower bound for a single item’s latency: \(\Delta t \geq \Delta t_U + \Delta t_C + \Delta t_D\). Throughput can however be improved from the default non-overlapping case \(f = 1/\Delta t\) significantly: The Tesla series GPUs for scientific computing offer dedicated hardware queues for upload and downloads, which increases the upper bound to \(f = 1/\max(\Delta t_U, \Delta t_C, \Delta t_D)\). The GeForce consumer models share a single copy engine but still raise the limit to \(f = 1/\max(\Delta t_C, \Delta t_U + \Delta t_D)\). The pattern which our scheduling follows by default in each iteration \(k\) is suitable for both types of GPUs:

1) pull work item \(k\)
2) enqueue upload of item \(k\)
3) enqueue download of item \(k - 1\)
4) enqueue computation of item \(k\)
5) push item \(k - 2\) after it has been downloaded
Exceptions are made to maintain low latencies: While only 5) uses blocking calls, we also try to push item \(k - 2\) and even \(k - 1\) to the next filter in a non-blocking way on earlier occasions. If a new item \(k\) does not become available before the computation of \(k - 1\) finishes, the latter’s download is given priority. This abandons overlap for one iteration, but prevents item \(k - 1\) from unnecessarily waiting for \(k\).

Please note that users’ implementations of the virtual Step BaseGPU::upload/compute/download() methods must not synchronize the GPU. This obviously excludes cudaDeviceSynchronize() but also GPU memory allocation. As an alternative, \textit{libHawaii} provides pools which internally organize memory in bins of \(2^i, i \in [8, 32]\) bytes.

2) Implicit Memory Management: The impact of discrete GPU memory on code complexity can be mitigated more easily than that on throughput. Due to \textit{libHawaii}’s origin in computer vision, we do so using \textit{OpenCV} types: Mat and GpuMat represent images as well as matrices in CPU or GPU memory respectively. By mimicking their interface, \textit{libHawaii}’s AutoMat can be used as a drop-in replacement. It stores one \textit{Mat} and an individual \textit{GpuMat} for each GPU, and keeps track of their validity – write access by the 2\textsuperscript{nd} GPU e. g. invalidates all other instances. On read access from a specific processor, data is copied from the quickest valid source: The ranking for a discrete GPU e. g. begins with its own memory, followed by that of peer-to-peer-accessible GPUs. Thereafter, page-locked is preferred to pageable CPU memory. Lastly, data from a non-peer-to-peer-accessible GPU is copied to the host and from there to the target GPU.

IV. EXPERIMENTAL EVALUATION

The following evaluation of \textit{libHawaii} employs two sample applications, which at first are introduced briefly. After that, the respective programming efforts for interfacing them with our library are presented in detail using, inter alia, exemplary source code excerpts. The performance and energy efficiency achieved by our automatic adaptation strategies are analyzed and compared to each applications’ baselines.

A. Sample Applications

Our aim in selecting the applications described below was to broaden the scope of our initial feasibility analysis [8]: Its single application of dense stereo vision combines various image processing operations and is computationally demanding, but also mostly deterministic by rarely depending on the input images’ actual content. The second sample application is contrary in this respect while the first one is very basic:

1) GEMM: General matrix multiplication is a fundamental subroutine for high-performance computing applications. The best-known among these is probably the \textit{LINPACK} [45] benchmark used to rank the world’s top 500 supercomputers [46]. Instead of only multiplying two matrices, GEMM computes the expression \(\alpha \mathbf{AB} + \beta \mathbf{C}\). This allows \(\mathbf{A}\) and \(\mathbf{B}\) to be sub-blocks of much larger matrices to be multiplied – the products of all such sub-blocks can then be found in a cache-friendly way before being accumulated to form a sub-block of the result matrix. Our sample application operates on real-valued matrices sized 1024 x 1024 elements in single-precision format. For heterogeneous pipelining, the above expression had to be split into two steps: a matrix multiplication \(\mathbf{T} = \mathbf{AB}\), followed by a scaled addition \(\alpha \mathbf{T} + \beta \mathbf{C}\). Our CPU implementation wraps the \textit{Eigen} library [47] for linear algebra and adds multi-core support via \textit{OpenMP}, while \textit{CUBLAS} [48] is used on GPUs – both have proven to be very efficient on their respective platforms.

2) Feature Matching: Like the previously evaluated dense stereo vision, feature matching is a computer vision application. It differs in one important respect however: Instead of providing dense results across most pixels, it only finds correspondences between small sets of points from different images\(^5\). These keypoints are selected by detecting corners or small blobs – their number and local distribution may therefore vary significantly depending on an image’s content. This causes irregular memory accesses during the subsequent process of describing their appearance.

The development of descriptors, i. e. concise representations of a keypoint’s appearance, is an active research topic. In addition to being efficiently computable and comparable, requirements include robustness against changes in illumination, rotation and scale. Open source implementations for both CPUs and GPUs exist for the \textit{SURF} [49] and \textit{ORB} [50] descriptors – nevertheless the former’s usage is restricted by patents [51]. We chose the latter for evaluating \textit{libHawaii}: \textit{ORB} has not been designed from scratch, but rather built upon \textit{BRIEF} [52] – both describe a keypoint’s appearance using a 256-bit vector containing the results of 256 pairwise brightness comparisons between pixels around it. This allows quantifying the dissimilarity of two descriptors by finding their \textit{Hamming} distance, i. e. the number of unequal bits – a computation suitable for the population count instructions of most modern processors. While comparing

\(^5\)Finding results for fewer pixels may be a disadvantage. In compensation, sparse feature matching can successfully be applied e.g. to images captured before and after considerable camera movement, while dense stereo vision usually requires rigidly-connected cameras and off-line calibration.
Their purely-virtual methods can be implemented more conveniently because their arguments use the actual rather than typedefs to "boost::shared_ptr< ItemGEMM >". They apply the largest remainder method, which is also common for the allocation of seats after elections. They are typedefs to "boost::shared_ptr< ItemFeatureMatching >". B. Programming Efforts

Instead of statistics on the lines of code needed to interface the sample applications with libHawaii, this section will present excerpts of that code itself. We believe this direct view is more insightful for potential users who want to estimate the same efforts w.r.t. their own applications. As described in section III-A, work items and processing steps are the building blocks from which application-specific types must be derived. Figs. 5 and 6 exemplarily illustrate this for the feature matching work item and the GPU-based GEMM processing step. The base class templates ItemTemplate and StepTemplate(GPU) extend their parents ItemBase and StepBase(GPU) by only one aspect: Their purely-virtual methods can be implemented more conveniently because their arguments use the actual rather than the base item type. The methods for partitioning the item are merely declared in fig. 5 because their definition is repetitive: Current keypoints and descriptors are split proportionally to the ratios argument, while previous ones are shared as a pointer. The original input image is divided into stripes which must overlap by the area within which the ORB descriptor compares pixels. Since this is common in computer vision, our library provides helper functions for mapping continuous partitioning ratios to discrete and optionally overlapping intervals. They apply the largest remainder method, which is also common for the allocation of seats after elections.
ARDWARE SPECIFICATIONS OF TEST SYSTEM

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5645</td>
<td>nVidia GeForce GTX 470</td>
</tr>
</tbody>
</table>

C. Performance and Energy Efficiency

Prior to quantitatively evaluating the performance and energy efficiency achieved by the different strategies, table I introduces our test system. We use such systems not only stationarily, but also in autonomous prototype vehicles [53].

Two experiments were conducted to analyze the optimization of performance and energy efficiency respectively. First, we provided input work items at an unlimited rate, making each adaptation strategy attempt to keep pace by maximizing throughput. Fig. 10 presents the performance achieved by each strategy and – as a baseline for comparison – by using all CPU cores or a single GPU only. In this context, “series” means sequential execution of processing steps for each item while each step leverages data parallelism. Two main observations can be made:

- GEMM fully supports asynchronous CUDA streams for overlapping GPU memory transfers and computations. Therefore, throughput could be improved by 76% on average, with the drawback of latency also increasing
by 59%. Both numbers were just 8% for our 2nd sample application, because its synchronous feature detection and extraction step constitutes the majority of runtime.

- Partitioning yielded lower throughputs than demand-based allocation, even though we verified that the found ratios represent a global optimum, i.e. their optimization operates successfully. Instead, both applications include measurable per-item overheads independent from their partition ratios. The higher latencies can however be explained by our experiment: Partitioning prepares a new set of work item partitions immediately after processing of the current set has started. While this reduces the latency of actual on-line applications, it practically implements a one-item queue when combined with our experiment’s unlimited input rate.

Even though energy efficiency is explicitly analyzed only by the following experiment, it has already been optimized here as well: Since the actual matching of features could easily keep pace with their more demanding detection and extraction step, the majority of runtime.

We conclude by revisiting the goals stated in section I: Concerning generality, we have demonstrated our library’s capability of exploiting applications’ data element, data flow and task parallelism by evaluating two considerably different sample applications. Its adaptation strategies have proven to successfully optimize not only performance but also energy efficiency continuously throughout runtime. The impact of transfers between a system’s main accelerators’ discrete memory has been effectively mitigated. These optimizations are portable to a different system merely by specifying its processors’ thermal design powers. Regarding productivity, the presented code examples allow an own assessment of libHawaii being worthwhile for own programs.

While optimization within the presented strategies occurs automatically, we did not automate the selection of strategies: It is considerably less tedious than the tuning of scheduling parameters and can follow few simple, system-independent but application-specific guidelines: Pipelining is preferable for processing steps that either perform IO or are exceptionally suitable/exclusively available for a specific processor. Using demand-based allocation and overlapping GPU memory transfers and computations maximizes throughput and therefore allows processing static data sets in the shortest time. It is also likely to offer the best efficiency. In contrast, partitioning should be employed by on-line applications for which latency is critical. Regarding applications, we will next apply libHawaii to a full-scale advanced driver assistance system of an autonomous prototype vehicle.