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 *libHawaii*[1]. 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

[1]This 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 source[2] *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 heterogeneously. 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 *PEPPHER* 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

[2]to become available under the terms of the *GNU General Public License* at our partner institute's website www.mrt.kit.edu/software. A non-public preview is provided at www.mrt.kit.edu/ranftweb/libHawaiiPreview.tar.gz.

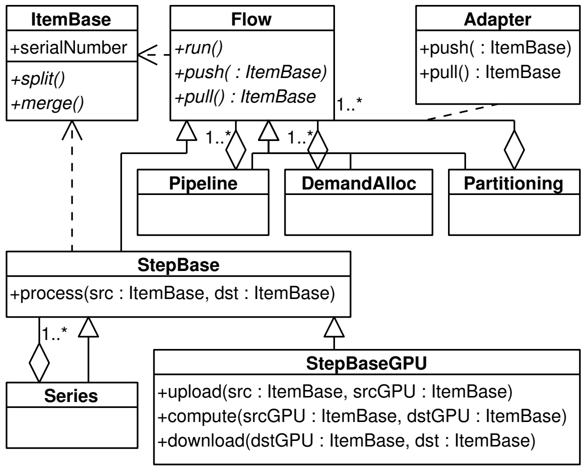

Figure 1. UML class diagram of *libHawaii* types related to adaptive heterogeneous computing, including only relevant methods and members

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 *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 using *CUDA*-capable GPUs in particular. We will refer to *libHawaii*'s UML class diagram in fig. 1 repeatedly.

A. Fundamental Building Blocks

Virtual inheritance is a common pattern not only within our library. Interfacing it with an application also requires deriving specific versions of `ItemBase` and `StepBase`:

1) Work Item: As stated above, we expect an application to process a stream of work items, which transport inputs as well as intermediate and final results between filters. Since this data is application-specific, it should be defined in one or more child classes of `ItemBase`. The base itself merely contains a running number for timing latencies and sorting items. The methods for splitting an input item into smaller parts and merging the respective results are only required for partitioning an item type across heterogeneous processors.

2) Processing Step: This class – often called a filter – applies computations to work items via its virtual method `StepBase::process()`. Any single processor can be used in it through an API of choice. Once an application is run, each instance of `StepBase` launches its own thread to repeatedly pull, process and push one work item at a time. Dividing applications into more rather than fewer steps ensures flexibility, while the `Series` class still allows to easily re-combine and to sequentially apply them.

Optimizing an application's energy efficiency depends on knowing the power consumption of every filter instance on its associated processor. Appropriate sensors are rare in ordinary systems and likely unable to assign a measurement to concurrent filters sharing one processor. *libHawaii* therefore contains a model which merely requires specifying the GPU or the number of CPU cores utilized while a step processes a work item. This time interval Δt is measured individually by each step instance[3]. We observed that processors spend most time in either their highest or idle voltage/frequency state[4], and thus assume that each processor i consumes its thermal design power (TDP) P_i while being utilized. Given this characteristic value for each processor of a heterogeneous system, a single filter instance's energy per work item can be estimated:

$$w = \sum_i P_i \, \Delta t_i \tag{1}$$

3) Base Flow Component: *libHawaii*'s interface to application code is now complete, but one fundamental building block remains: As fig. 1 indicates, `Flow::pull()` and `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 f based on noisy measurements $\hat{f}$ acquired from the k^{th} work item:

$$f_k = \alpha \, f_{k-1} + (1-\alpha) \, \hat{f}_k, \; \alpha \in [0,1) \tag{2}$$

Apart from these measurements, an instance of `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: `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 *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.

[3]using *Linux*'s `getrusage()` call for CPUs and the event API of GPUs

[4]from *Linux*'s file `/sys/devices/system/cpu/cpu*/cpufreq/stats/time_in_state` for CPUs and the `nvprof` profiler for GPUs

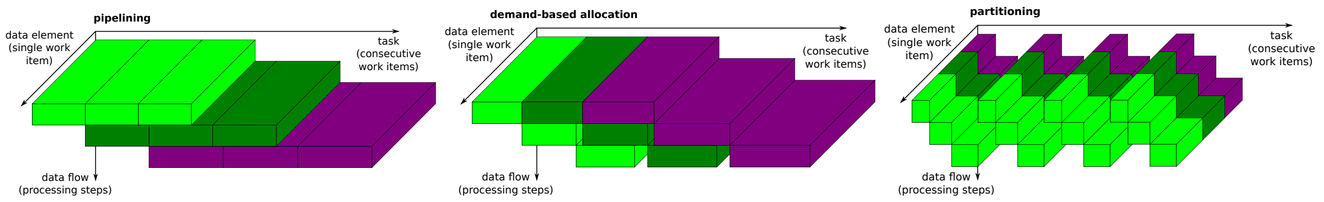
pipelining
data element (single work item)
task (consecutive work items)
data flow (processing steps)
demand-based allocation
data element (single work item)
task (consecutive work items)
data flow (processing steps)
partitioning
data element (single work item)
task (consecutive work items)
data flow (processing steps)

Figure 2. Strategies to exploit different kinds of parallelism using multiple heterogeneous processors, each being indicated by an individual color

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_{min} required to keep pace with the stream of input items is measured. Based on the processors' unconstrained throughputs $f_{unc,i}$ the j most efficient ones can be spared from any throughput limit f_{lim}:

$$f_{lim,j} = \infty \; \forall j : \sum_{i=1}^{j} f_{unc,i} < f_{min} \tag{3}$$

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

$$f_{lim,j+1} = f_{min} - \sum_{i=1}^{j} f_{unc,i} \tag{4}$$

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_{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:

$$1/f = m\,r + c \tag{5}$$

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 $\mathbf{H}_k$. Its state vector $\hat{\mathbf{x}}_k$ 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:

$$1/f_k = \mathbf{H}_k\,\hat{\mathbf{x}}_k = \begin{pmatrix} r_k & 1 \end{pmatrix} \begin{pmatrix} m_k & c_k \end{pmatrix}^\intercal \tag{6}$$

A parameter update requires several helper variables: The internal constants $\mathbf{Q}$ and R define the noise (co-)variance of the state vector and the period respectively. The state error co-variance $\mathbf{P}_k$ measures the uncertainty of the most recently estimated model parameters. Finally, the *Kalman* gain $\mathbf{K}_k$ 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 $1/f_k$:

$$\mathbf{K}_k = \mathbf{P}_{k-1}\mathbf{H}_k^\intercal \left(\mathbf{H}_k\mathbf{P}_{k-1}\mathbf{H}_k^\intercal + R\right)^{-1} \tag{7}$$
$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} + \mathbf{K}_k\left(1/f_k - \mathbf{H}_k\,\hat{\mathbf{x}}_{k-1}\right) \tag{8}$$
$$\mathbf{P}_k = \left(\mathbf{I} - \mathbf{K}_k\mathbf{H}_k\right)\mathbf{P}_{k-1} + \mathbf{Q} \tag{9}$$

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_{min} in an energy-efficient way by first finding the maximum allowed work item ratio of each processor:

$$r_i = \left(1/f_{min} - c_i\right)/m_i \tag{10}$$

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:

$$\begin{pmatrix} m_1 & -m2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & m_{n-1} & -m_n \\ 1 & 1 & \cdots & 1 & 1 \end{pmatrix} \begin{pmatrix} r_1 \\ \vdots \\ r_{n-1} \\ r_n \end{pmatrix} = \begin{pmatrix} c_2 - c_1 \\ \vdots \\ c_n - c_{n-1} \\ 1 \end{pmatrix} \tag{11}$$

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.

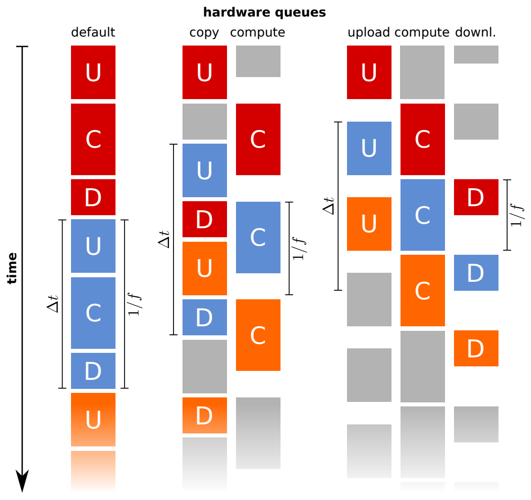

Figure 3. Time-lines of uploads (U) to, computations (C) on and downloads (D) from a discrete-memory GPU: default scheduling (left), pipelined processing with one shared copy engine (center) and with dedicated up- and download engines (right). Δt is latency, $1/f$ inverse throughput or period.

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 up- 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 `StepBaseGPU::upload/compute/download()` methods must not synchronize the GPU. This obviously excludes `cudaDeviceSynchronize()` but also GPU memory allocation. As an alternative, *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 *libHawaii*'s origin in computer vision, we do so using *OpenCV* types: `Mat` and `GpuMat` represent images as well as matrices in CPU or GPU memory respectively. By mimicing their interface, *libHawaii*'s `AutoMat` can be used as a drop-in replacement. It stores one `Mat` and an individual `GpuMat` for each GPU, and keeps track of their validity – write access by the 2nd 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 *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 *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

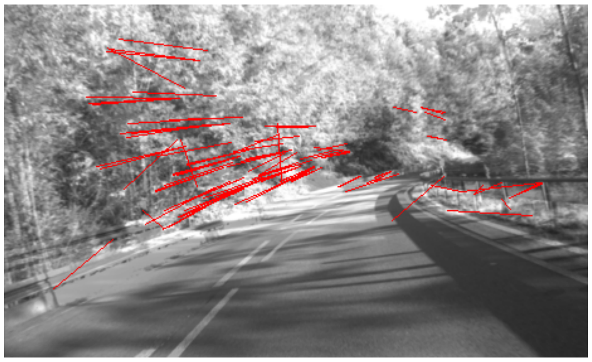

Figure 4. *ORB* feature matches between overlaid consecutive images from a motorcycle-mounted camera: few results on ground due to motion blur

– 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 *Eigen* library [47] for linear algebra and adds multi-core support via *OpenMP*, while *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 *SURF* [49] and *ORB* [50] descriptors – nevertheless the former's usage is restricted by patents [51]. We chose the latter for evaluating *libHawaii*: *ORB* has not been designed from scratch, but rather built upon *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 *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.

```
// derive work item for feature matching using the "curiously recurring template pattern"
class ItemFeatureMatching : public hawaii::flow::ItemTemplate< ItemFeatureMatching,
                                                              hawaii::flow::PartInfoImageOverlap > {
  // inputs, intermediate and final results
  public:
  hawaii::AutoMat             imageCurr,       imagePrev       ;
  std::vector< cv::KeyPoint > keypointsCurr,   keypointsPrev   ;
  hawaii::AutoMat             descriptorsCurr, descriptorsPrev ;
  hawaii::AutoMat             matchIndices, matchDistances ;

  // split and re-combine work item (required for partitioning only)
  public:
  virtual void splitImpl(       std::vector< ItemFeatureMatching::Ptr >& parts,
                          const std::vector< double >&                   ratios,
                          const hawaii::flow::PartInfoImageOverlap&      partInfo ) ;
  virtual void mergeImpl( const std::vector< ItemFeatureMatching::Ptr >& parts,
                                ItemFeatureMatching::Ptr&                whole,
                          const hawaii::flow::PartInfoImageOverlap&      partInfo ) ;
} ;
```

Figure 5. Work item type for feature matching: Not deriving directly from ItemBase, but instead via its child class template ItemTemplate, merely makes implementing the partitioning methods more convenient – their arguments would otherwise need to be cast from and to ItemBase pointers.

brightnesses already compensates for illumination changes, *ORB* adds rotation- and scale-invariance by adapting the distance and angle between a keypoint's center and each adjacent pixel to be compared with one another.
As shown in fig. 4, our sample application matches corresponding features, i. e. tuples of a keypoint and its descriptor, between consecutive frames of a video stream. Such matches are useful e. g. for estimating the camera's motion, but also create a dependency between subsequent work items. The next section explains how such dependencies can be made compatible with heterogeneous parallel processing. For the sake of simplicity and because algorithmic improvements of computer vision are not in the focus of this paper, our matching processing step is very straightforward: For each of the current image's features, the corresponding match from the previous image is selected by minimizing their *Hamming* distance. Data-parallel implementations of this algorithm as well as the aforementioned detection and extraction of features for CPU and GPU are included in *OpenCV* [11]. Unfortunately, the first of its corresponding classes ORB_GPU and BFMatcher_GPU does not support enqueuing its operations to any but the default stream. This unnecessarily synchronizes them with all other tasks on that stream, particularly feature detection and extraction on a different GPU.

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 whole. 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.

```
// derive GEMM processing step for one GPU
class GEMMGPU :
  public hawaii::flow::StepTemplateGPU< ItemGEMM > {

  // options as members, CUBLAS support
  public:
  float alpha, beta ;
  protected:
  cublasHandle_t handle ;

  // implement purely-virtual methods: "{Src,Dst}(GPU)Ptr"
  // are typedefs to "boost::shared_ptr< ItemGEMM >".
  // always pass the same work item through
  public:
  void uploadImpl( SrcPtr&          srcPtr,
                   SrcGPUPtr&       srcGPUPtr,
                   cv::gpu::Stream& stream    ) const {
    srcGPUPtr = srcPtr ;
    srcGPUPtr->A.readGPU( stream ) ;
    srcGPUPtr->B.readGPU( stream ) ;
    srcGPUPtr->C.readGPU( stream ) ; }
  void computeImpl( SrcGPUPtr&       srcGPUPtr,
                    DstGPUPtr&       dstGPUPtr,
                    cv::gpu::Stream& stream    ) const {
    dstGPUPtr = srcGPUPtr ;
    gemmGPU( srcGPUPtr->A, srcGPUPtr->B, srcGPUPtr->C,
             dstGPUPtr->result,
             this->alpha, this->beta,
             this->handle, stream ) ; }
  void downloadImpl( DstGPUPtr&       dstGPUPtr,
                     DstPtr&          dstPtr,
                     cv::gpu::Stream& stream    ) const {
    dstPtr = dstGPUPtr ;
    dstPtr->result.readCPU( stream ) ; }
} ;
```

Figure 6. GPU Processing step for general matrix multiplication: Separate methods for uploading inputs, performing computations and downloading results allow overlapping these operations on consecutive work items.

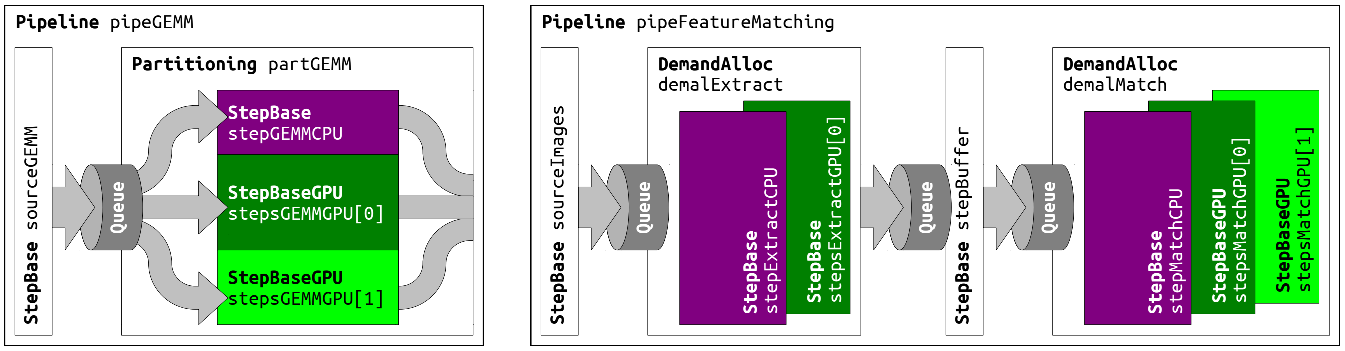

Figure 7. Illustrated configurations for adaptive heterogeneous computing as listed in figs. 8 (left) and 9 (right): The system has a number of symmetric multiprocessing CPU cores as well as two GPUs. While GEMM only receives its input items through a pipeline, feature matching between consecutive images also requires an intermediate buffer for the previous keypoints and descriptors – its processing steps must therefore be optimized separately. Feature detection and extraction cannot effectively use the 2nd GPU because its wrapped *OpenCV* code merely enqueues to the synchronous default *CUDA* stream.

```
// adaptive partitioning across all CPU cores and GPUs
hawaii::flow::Partitioning partGEMM( stepGEMMCPU ) ;
for( int GPU = 0 ; GPU < hawaii::GPUs ; ++GPU ) {
  partGEMM.add( stepsGEMMGPU[ GPU ] ) ;
}

// add source of input matrices via pipeline,
hawaii::flow::Pipeline pipeGEMM(           sourceGEMM,
  hawaii::flow::queueFactory( typeFIFO ), partGEMM   ) ;

// run for 30 seconds, then print evaluation
pipeGEMM.run() ;
sleep( 30 ) ;
pipeGEMM.eval() ;
pipeGEMM.stop() ;
```

Figure 8. Use of partitioning for GEMM: `partGEMM` receives new work items from `sourceGEMM` through a FIFO queue. Since generating new input matrices is much faster than multiplying them, `partGEMM` attempts to keep pace by optimizing performance rather than energy efficiency.

The final step towards implementing adaptive heterogeneous computing is combining one's filters within one or more of the strategies introduced in section III-B. As before, this is demonstrated by source code excerpts – fig. 7 provides complementary illustrations of the example configurations to be described: Both employ a pipeline to connect their IO-based work item sources to their respective processing. Fig. 8 explains how all processors of a heterogeneous system can be configured to cooperate at GEMM using the partitioning strategy. Aside from showing `DemandAlloc`'s identical interface, fig. 9 presents a solution to the dependency between consecutive feature matching items: Because filters and adaptation strategies may be flexibly nested, the extraction and matching steps can be optimized independently – even by different strategies. This allows inserting a queue to sort incoming work items and a buffer to add the respective previous features to each current item in between.

Table I
HARDWARE SPECIFICATIONS OF TEST SYSTEM

	CPU	GPU
model	Intel Xeon E5645	nVidia GeForce GTX 470
count	2 sockets	2 expansion cards
parallelism	6 cores x 128-bit SIMD	14 streaming multiprocessors x 32 SIMT cores
clock frequency	2400 MHz	607 MHz
memory bandwidth	32.0 GB/s	133.9 GB/s

```
// demand-based allocation: OpenCV's wrapped feature
// detection/extraction effectively supports one GPU only.
hawaii::flow::DemandAlloc demalExtract( stepExtractCPU ) ;
hawaii::flow::DemandAlloc demalMatch(   stepMatchCPU   ) ;
if( hawaii::GPUs > 0 ) {
  demalExtract.add( stepsExtractGPU[ 0 ] ) ;
}
for( int GPU = 0 ; GPU < hawaii::GPUs ; ++GPU ) {
  demalMatch.add( stepsMatchGPU[ GPU ] ) ;
}

// add source of input images and buffer for previous
// keypoints/descriptors via pipeline
hawaii::flow::Pipeline pipeFeatureMatching(  sourceImages,
  hawaii::flow::queueFactory( typeFIFO    ), demalExtract,
  hawaii::flow::queueFactory( typeOrdered ), stepBuffer,
  hawaii::flow::queueFactory( typeFIFO    ), demalMatch
) ;

// emulate real camera's frame rate of 30 Hz
sourceImages.setThroughputLimit( 30.0 ) ;

// run until the final image has been processed
pipeFeatureMatching.run() ;
pipeFeatureMatching.wait() ;
```

Figure 9. Use of demand-based allocation for feature matching: If `demalExtract` exceeds the image source's throughput of 30 Hz, its performance is sufficient to allow optimizing energy efficiency. The same logic applies to `demalMatch` if it can keep pace with `demalExtract`.

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

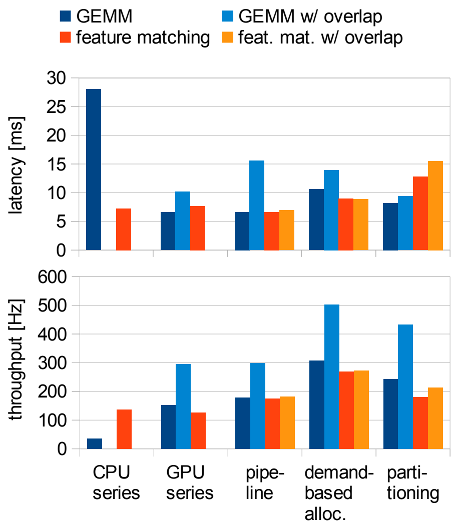

Figure 10. Latency (top) and throughput (bottom) achieved by each combination of scheduling and application: Apart from *libHawaii*'s adaptation strategies, scheduling includes all CPU cores and a single GPU as references. If available, both sample applications have additionally been profiled with overlapping GPU memory transfers and computations.

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, `DemandAlloc` and `Partitioning` were already able to prefer the more efficient GPUs for this step.

In our second experiment, we limited the rate of input items to trigger adaptive energy efficiency optimization. We arbitrarily chose exactly half the throughput achieved before by each respective combination of scheduling strategy and sample application. As fig. 11 shows, single processors and the non-adaptive pipelining strategy on average required 55% of their original estimated power consumption to maintain this throughput, while the adaptive strategies only used 45%. This was realized differently for each application: At feature detection and extraction, the more efficient CPUs still needed 30% of support by a GPU. Basic GEMM preferred both GPUs equally, while even a single GPU was found to be most efficient and fast enough if overlapping is enabled.

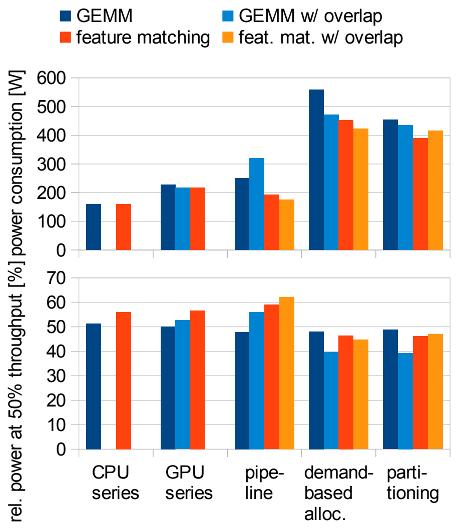

Figure 11. Absolute power consumption while achieving maximum performance (top) and relative power consumption (bottom) when limiting the input item rate to 50% of each respective maximum: Only demand-based allocation and partitioning can explicitly prefer more efficient processors.

To finally quantify the overheads introduced by wrapping an application's code in *libHawaii* classes, we repeated the above experiments with empty processing steps: Overall latencies did not exceed 10 µs, so overheads are negligible for application throughputs below at least 1000 Hz.

V. Conclusions and Outlook

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 and 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.

REFERENCES

[1] D. Delling *et al.*, “Phast: Hardware-accelerated shortest path trees,” in *IPDPS*, 2011.

[2] B. Ranft *et al.*, “Parallel matching-based estimation - a case study on three different hardware architectures,” in *IV*, 2011.

[3] R. Vuduc *et al.*, “On the limits of gpu acceleration,” in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, 2010.

[4] What is heterogeneous system architecture (hsa)? Advanced Micro Devices, Inc. [Online]. Available: developer.amd.com/resources/heterogeneous-computing

[5] Opencl: the advantages of heterogeneous approach. Intel Corporation. [Online]. Available: software.intel.com/en-us/articles/opencl-the-advantages-of-heterogeneous-approach

[6] big.little processing. ARM Ltd. [Online]. Available: arm.com/products/processors/technologies/biglittleprocessing.php

[7] T. Mattson *et al.*, *Patterns for parallel programming*, 1st ed. Addison-Wesley, 2004.

[8] B. Ranft and O. Denninger, “Run-time adaptation to heterogeneous processing units for real-time stereo vision,” in *HPCC-ICESS*, 2012.

[9] J. Hoberock and N. Bell. Thrust – parallel algorithms library. [Online]. Available: thrust.github.io

[10] Nvidia performance primitives. NVIDIA Corporation. [Online]. Available: developer.nvidia.com/npp

[11] Opencv. Itseez. [Online]. Available: opencv.org

[12] The openacc application programming interface. OpenACC. [Online]. Available: openacc.org/Downloads

[13] V. K. Elangovan *et al.*, “Ompss-opencl programming model for heterogeneous systems,” in *LCPC*, 2012.

[14] L. Chen *et al.*, “Dynamic load balancing on single- and multi-gpu systems,” in *IPDPS*, 2010.

[15] J. Kim *et al.*, “Snucl: an opencl framework for heterogeneous cpu/gpu clusters,” in *ICS*, 2012.

[16] A. Hormati *et al.*, “Sponge: portable stream programming on graphics engines,” in *ASPLOS*, 2011.

[17] C. Dubach *et al.*, “Compiling a high-level language for gpus: (via language support for architectures and compilers),” in *PLDI*, 2012.

[18] S. Kamil *et al.*, “An auto-tuning framework for parallel multicore stencil computations,” in *IPDPS*, 2010.

[19] Y. Ogata *et al.*, “An efficient, model-based cpu-gpu heterogeneous fft library,” in *IPDPS*, 2008.

[20] C.-K. Luk *et al.*, “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *MICRO*, 2009.

[21] T. Scogland *et al.*, “Heterogeneous task scheduling for accelerated openmp,” in *IPDPS*, 2012.

[22] T. D. R. Hartley *et al.*, “Automatic dataflow application tuning for heterogeneous systems,” in *HiPC*, 2010.

[23] V. T. Ravi and G. Agrawal, “A dynamic scheduling framework for emerging heterogeneous systems,” in *HiPC*, 2011.

[24] U. Dastgeer *et al.*, “Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems,” in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, 2011.

[25] M. Steuwer *et al.*, “Skelcl - a portable skeleton library for high-level gpu programming,” in *IPDPS Workshops*, 2011.

[26] M. Aldinucci *et al.*, “Targeting heterogeneous architectures via macro data flow,” *Parallel Processing Letters*, vol. 22, no. 2, 2012.

[27] P. Alonso *et al.*, “Tools for power-energy modelling and analysis of parallel scientific applications,” in *ICPP*, 2012.

[28] S. Song *et al.*, “A simplified and accurate model of power-performance efficiency on emergent gpu architectures,” in *IPDPS*, 2013.

[29] G. Wang and Y. Lin, “Heterogeneity-aware peak power management for accelerator-based systems,” in *ICPADS*, 2011.

[30] T. Hamano *et al.*, “Power-aware dynamic task scheduling for heterogeneous accelerated clusters,” in *IPDPS*, 2009.

[31] A. Benoit *et al.*, “Performance and energy optimization of concurrent pipelined applications,” in *IPDPS*, 2010.

[32] Threading building blocks. Intel Corporation. [Online]. Available: threadingbuildingblocks.org

[33] G. F. Diamos and S. Yalamanchili, “Harmony: an execution model and runtime for heterogeneous many core systems,” in *HPDC*, 2008.

[34] G. Teodoro *et al.*, “High-throughput analysis of large microscopy image datasets on cpu-gpu cluster platforms,” in *IPDPS*, 2013.

[35] M. Maheswaran *et al.*, “Dynamic mapping of a class of independent tasks onto heterogeneous computing systems,” *J. Parallel Distrib. Comput.*, vol. 59, no. 2, 1999.

[36] H. Topcuoglu *et al.*, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, 2002.

[37] S. Benkner *et al.*, “Peppher: Efficient and productive usage of hybrid computing systems,” *IEEE Micro*, vol. 31, no. 5, 2011.

[38] C. Augonnet *et al.*, “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, 2011.

[39] N. Khammassi *et al.*, “Mhpm: Multi-scale hybrid programming model: A flexible parallelization methodology,” in *HPCC-ICESS*, 2012.

[40] J.-J. Chen and L. Thiele, “Energy-efficient task partition for periodic real-time tasks on platforms with dual processing elements,” in *ICPADS*, 2008.

[41] Y. Li *et al.*, “A heuristic energy-aware scheduling algorithm for heterogeneous clusters,” in *ICPADS*, 2009.

[42] L. Wang *et al.*, “Power aware scheduling for parallel tasks via task clustering,” in *ICPADS*, 2010.

[43] M. Quigley *et al.*, “ROS: an open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009.

[44] G. Welch and G. Bishop, “An introduction to the kalman filter,” Chapel Hill, NC, USA, Tech. Rep., 1995.

[45] A. Petitet *et al.* Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. [Online]. Available: netlib.org/benchmark/hpl

[46] H. Meuer *et al.* Top500 supercomputer sites. [Online]. Available: top500.org

[47] B. Jacob *et al.* Eigen. [Online]. Available: eigen.tuxfamily.org

[48] Cublas. NVIDIA Corporation. [Online]. Available: docs.nvidia.com/cuda/cublas

[49] H. Bay *et al.*, “Surf: Speeded up robust features,” in *ECCV (1)*, 2006.

[50] E. Rublee *et al.*, “Orb: An efficient alternative to sift or surf,” in *ICCV*, 2011.

[51] R. Funayama *et al.*, “Robust interest point detector and descriptor,” Patent US 2 009 238 460, Sep. 24, 2009.

[52] M. Calonder *et al.*, “Brief: Binary robust independent elementary features,” in *ECCV (4)*, 2010.

[53] A. Geiger *et al.*, “Team annieway’s entry to the 2011 grand cooperative driving challenge,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 3, 2012.