CPU-GPU Heterogeneous Computing

Steffen Lammel
Institute of Computer Engineering (ZITI)
Ruprecht-Karls-Universitaet Heidelberg
S.Lammel@stud.uni-heidelberg.de

Abstract—This paper gives a brief overview of the latest CPU-GPU heterogeneous computing systems (HCS) and techniques. To achieve the best performance with these systems, both processing units (PUs) need to be utilized. In general, the higher utilization results also in a better energy efficiency compared to the CPU/GPU-only approach. There are solutions in software as well as hardware which contribute to this goal. Software is used to minimize idle-times of PUs and increase the overall performance. Fused systems (fused HCS) combine both CPU and GPU on a single chip and have benefits over discrete systems in certain cases. Hardware techniques like dynamic voltage-frequency scaling lower the overall power consumption of CPUs and GPUs in every operational state of the device.

Index Terms—Heterogeneous Computing, CPU, GPU, Accelerator, Energy-Efficiency

1 Introduction

The usage of a graphics processing unit (GPU) to accelerate an application or computing system has been a standard approach for several years. Many systems in the Top500 and Green500 rankings are equipped with GPU accelerators. Especially the top ranks of the Green500 are populated with systems that use GPUs as accelerators. This shows that GPUs offer a good performance per watt ratio and contribute to an overall increased energy efficiency of a computing system.

GPUs, however, are only one part of a typical computing system. To get as close to the theoretical peak performance of the system as possible, the CPU has to be utilized in a useful way, too, instead of merely acting as a supervisor for the GPUs. Combining the different paradigms and programming models of CPUs and GPUs to work collaboratively at the same problem is one great challenge. Another problem we face, is the distribution of the data. It is rather costly to transfer data from one processing unit to another. These data transfers are usually done via PCI-Express (PCIe) which is, compared to the DRAM of a CPU/GPU, very slow. Techniques which reduce these costly data movements to a minimum are therefore required.

1.1 Characteristics

Although CPUs and GPUs are in principle capable of doing the same calculations, there are huge differences in architecture, execution model and other details. Table 1 shows a brief comparison of a state-of-the-art CPU and GPU. The execution units differ largely on both devices. CPUs have a few, big, flexible, fast-clocked cores, whereas GPUs offer thousands of smaller, slower cores with reduced capabilities. The CPU uses its large cache-hierarchies to hide the latency of the memory (DDR3/DDR4). The GPU uses its caches to coalesce accesses to the memory to take advantage of the inherent speed of the memory (GDDR5/HBM).

In general, CPUs are better suited for latency-oriented tasks whereas GPUs perform better in throughput-oriented domains.

TABLE 1
CPU/GPU Comparison

	CPU (Intel Haswell)	GPU (Nvidia Kepler)
Cores	< 20	> 1000
Frequency	$\approx$ 3GHz	$\approx$ 1GHz
Caches	$<$ 30MB	$<$ 2MB
Memory (capacity)	$\leq$ 1TB	$\leq$ 12GB
Memory (speed)	$<$ 12GB/s	$<$ 250GB/s

1.2 Energy Efficiency

The great computational power of a GPU comes at almost the same energy cost of a CPU. The Intel Xeon E7 CPU delivers about 100 GFlops/s at the cost of about 150 Watts heat dissipation. The Nvidia GK110 (Kepler) GPU however, delivers 1.3 TFlops/s within a power budget of 250 Watts.

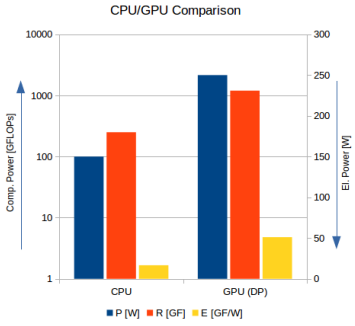


Fig. 1. Energy and efficiency comparison: P is the electrical power in watts; R is the computational power in GF/s; E is the efficiency in GF/s per watt

As we can see from these numbers which are shown in comparison in Fig. 1, a high utilization of the GPU is highly desirable. Unfortunately, this is only the case in synthetic benchmarks which favour the GPU. In real-world applications, it is very hard to utilize the GPU to exploit its maximum potential.

2 Workload Division Techniques

In order to make use of the potential of both processing units, the work needs to be divided. This needs to happen in an intelligent way, so that both processing units (PU) can contribute. A first approach would be a a partitioning scheme similar to PCAM [4]. With PCAM (partition, communicate, agglomerate, map), the whole problem will be split up into smaller sub-tasks. These sub-tasks should be as fine-granular as possible. If a sub-task can't be divided any further, the maximum amount of granularity is reached. Fig. 2 shows the basic idea of this concept.

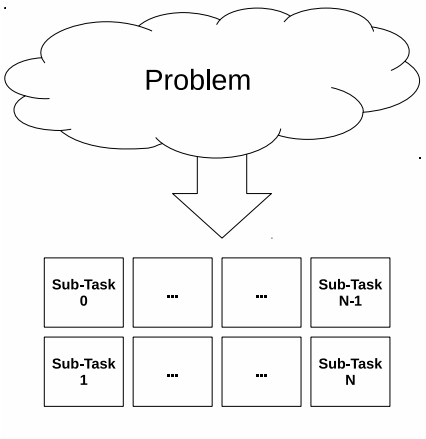


Fig. 2. Schematic view of the idea to divide a large problem into small sub-tasks

These fine grained tasks can then either be mapped to the CPU or the GPU.

2.1 Naive Partitioning

A very simple approach would be, to split the fine-grained tasks to all processing units by their presence in the system. E.g. if we have a CPU with two cores and a GPU which is acting as an accelerator, we map $\frac{2}{3}$ of the tasks to the CPU and $\frac{1}{3}$ to the GPU.

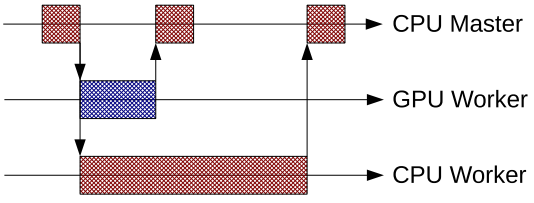


Fig. 3. Idle periods for the naive partitioning approach. Boxes represent work, empty lines are idle-phases

Unfortunately, it is very likely to have large idle-periods in the system with this approach. As shown in Section 1.1, the performance of CPUs and GPUs differs largely. In our artificial example we can assume that the GPU is much faster than the dual-core CPU. This will result in long idle periods for the GPU, which is what we want to avoid. Fig. 3 shows this issue. All the unoccupied parts in the timeline contribute to the idle time of the application and are not wanted at all.

2.2 Partitioning by the PU's relative performance

If we take the relative performance of the given PUs into account we can distribute the tasks more evenly balanced. If we know that our GPU is twice as fast as the CPU, we can therefore occupy the GPU with the double amount of work.

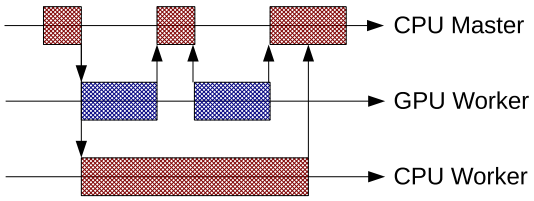


Fig. 4. Idle periods for the relative performance based approach

As Fig. 4 shows, the idle-periods are now reduced compared to the naive approach. To realize such an "X to Y" partitioning scheme, we need to know the performance of the given PUs before we dispatch the work. The raw-performance values which can be obtained from the devices data sheet is in the most cases not sufficient. These numbers only suggest how well a PU performs in a very specific domain which is especially suited for this PU. To gather some knowledge about how well our processing units perform on the given problem, some sort of micro-benchmark is required which tells us beforehand how well each PU performs on the specific problem.

2.3 Partitioning by nature of the sub-tasks

The quantitative distribution of the tasks is an improvement over the naive approach. A further improvement would be, to identify the task which are especially suited for a given PU and map those task exclusively to the respective PU. As shown in Section 1.1, in general a CPU is better suited for latency-critical tasks, while the GPU is better when throughput is heavily required.

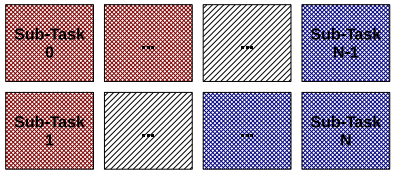


Fig. 5. Partitioning by the sub-task's characteristics: Red tiles are CPU-affine, blue tiles are GPU-affine. Grey tiles do not prefer any PU.

Fig. 5 extends the example shown in Fig. 2 with these assumptions. The red boxes show tasks which perform well

on a CPU. The blue boxes show tasks which perform well on a GPU. The grey boxes represent neutral tasks which execute equally well on both.

There are certain metrics which can be utilized to characterize the task and determine if it is rather CPU- or GPU-affine.

- Memory footprint
- Data Order/Access Patterns
- BLAS-Level

The memory footprint of the application is the most obvious metric. If the application requires more memory than the PU offers, it can not be executed on this PU (or only partially). GPUs are very limited in this regard whereas CPUs can be equipped with plenty of RAM. Irregular data patterns which cause a lot of random accesses are better suited for CPUs and their large, fast caches. If the data is ordered, the GPU can fully utilize its fast memory and highly parallel execution units. The amount of certain BLAS (basic linear algebraic subroutines) operations is also an indicator for the PU affinity. Vector and Vector-Matrix Operations (BLAS Level 1 and 2) are operations that perform well on CPU. Matrix-Matrix operations (BLAS Level 3) are better suited for GPUs.

2.3.1 Scheduling

With this approach, task-scheduling becomes a concern. In the worst case, we have many tasks which compete for execution on a specific PU. The following metrics should be respected when it comes to scheduling a task:

- Locality (of the data)
- Criticality (of the task)
- Availability (of the PU)

If a task is currently located on a CPU, but the characterisation as shown in Fig. 5 deemed the task to be better run on a GPU, we have to consider to run it on the CPU anyway. On high contention, the extra data transfer into the GPU's address space might unnecessarily stall the execution of other tasks whose data is already in-GPU. The same applies to tasks which are in the "wrong" memory space, but are uncritical compared to other tasks. The intentional execution on the non-optimal PU can be beneficial to the whole application. If a PU is completely unavailable, the task also needs to be scheduled on the non-optimal PU.

2.4 Pipeline

Another method to place the tasks to the PUs is the pipeline. If the tasks are dependent on each other, the pipeline is a viable approach to utilize all PUs.

The schematic pipeline shown in Fig. 6 consists of three stages, where every stage is run on a specific PU. Each subtask is dependent on the previous stage (except the first stage, of course). When the first stage, has finished the first part of Task A, it hands over the associated data to the next stage and can begin immediately with the first part of Task B. The second stage behaves the same way, except for a short waiting period for the first data. The third stage is the last one in our example and therefore responsible for saving the result. Li et al. [6] describe a way to multiply matrices

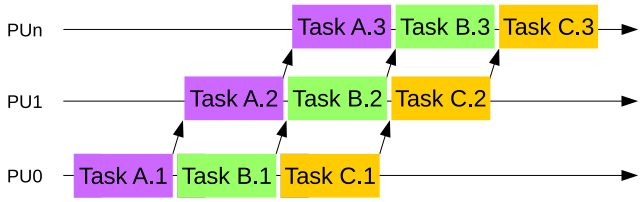


Fig. 6. Schematic view of a three-staged pipeline. With this pipeline, up to three tasks can be processed concurrently

(DEGMM) which are too big for the graphic card's memory partially in a pipelined fashion.

In principle, the pipeline can become as deep as the computing system's number of processing units. With deeper pipelines, the initial latency grows larger. After this filling period, all PUs are used. Large workloads which keep the pipeline occupied for a long time are preferable. The fill- and drain phases become negligible in this way. With this approach the nature of the sub-tasks (as discussed in section 5) is important again.

3 Frameworks

The techniques discussed in Section 2 are basics for almost every CPU-GPU heterogeneous computing system. Manual implementation with CUDA/OpenCL semantics only can be tedious and error prone though. To ease the programming effort, researchers propose different frameworks which are meant to solve these tasks in an automated way. These frameworks shall support programmers in different ways:

- Load balancing
- Parallel abstraction

With a framework dedicated to load-balancing, the task-mapping techniques described in Section 2 are handled by a framework. The scheduling can either be static or dynamic. With a static scheduler, the mapping-decisions for each PU is happening before the runtime of the application. Dynamic scheduling takes the mapping decisions to the runtime of the application. This enables additional degrees of freedom as the scheduling behavior can be adjusted during the runtime. The scheduler is able to gather information from the currently running tasks and use this informations later for upcoming scheduling decisions.

A framework dedicated to parallel abstraction is supposed to help the programmer write optimized code for heterogeneous environments. The optimal case would be to write a strictly sequential program and let the framework translate and optimize the code for an environment with multiple CPUs and GPUs. Beside such "fully-automatic" frameworks, there are approaches which take the burden away form the programmer only partially. These solutions require program annotations which tell the framework/compiler which parts of the code are suitable for a specific PU. Such solutions are very similar to OpenMP and its directives (e.g. *#pragma omp parallel*).

Mittal et al. [1] has collected many resources about such frameworks. Citing all of them would blow this report out of proportion, therefore two exemplary techniques have been selected to be shown in further detail.

3.1 SnuCL

SnuCL by J. Kim et al. [2] is an OpenCL framework for heterogeneous CPU/GPU cluster computers and targets high performance and ease of programming. This framework transforms an OpenCL application written for one node, to be run on many independent nodes. These nodes are connected with common networks like Gigabit Ethernet, 10 GbE or InfiniBand.

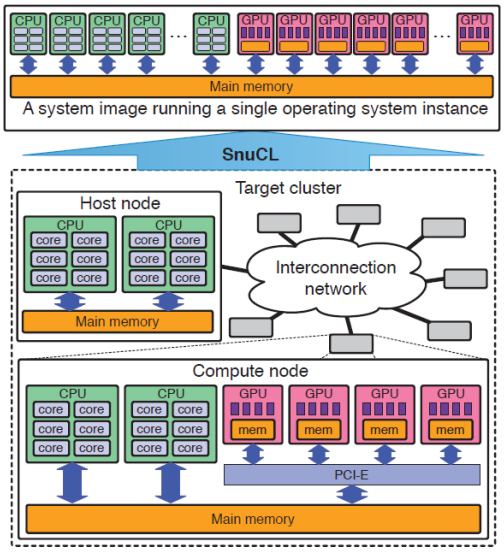


Fig. 7. SnuCL framework, overview [2]

The OpenCL code is mapped to a virtual super node which represents all PUs available in the cluster. SnuCL then transforms the code for this super node to all real nodes. Communication and data exchange with distant nodes is realized via a message passing interface (MPI). This inter-node communication is implicit. The programmer does not have to deal with MPI semantics as the translation from OpenCL data movements to MPI is handled completely by the framework.

3.1.1 Performance

The authors of SnuCL have used the NAS parallel benchmark [8] for performance evaluation. They compare the code generated by the framework with the reference implementation of the benchmark. They show that they achieve speed-ups compared to one CPU. Fig. 8 shows the speed-ups for increasing numbers of PUs (Single-CPU vs Multi-CPU, Multi-GPU, Multi-CPU/GPU).

The workload, however, was embarrassingly-parallel. This suggest that good performance depends largely on suitable data-structures.

3.2 PLASMA

PLASMA by S. Pai et al. [3] is a framework which provides code-abstraction for multiple accelerator architectures. Besides GPUs, it supports the CPUs SIMD extensions, the Cell

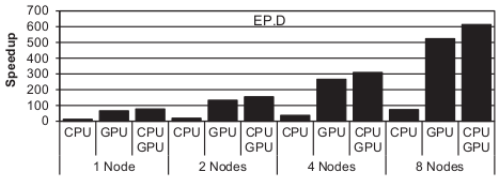


Fig. 8. SnuCL Performance [2]: Utilization of CPU only, GPU only and both, scaled over several nodes

Processor and some other architectures. The goal of this framework is to enable the creation of portable, platform independent code for heterogeneous systems. Fig. 9 shows a visualization of the frameworks general design. Fig. 10 shows the compilation process for multiple accelerator architectures.

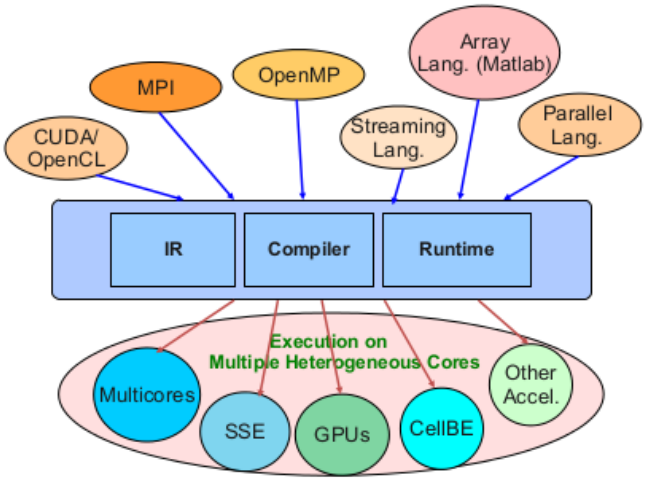


Fig. 1. PLASMA Overview

Fig. 9. PLASMA Overview - supported languages and accelerators

The main component is an intermediate representation (IR) which provides a clean abstraction without any platform specific SIMD dependencies. Specific optimizations for the used PUs are introduced when it comes to the compilation for the respective PU. Furthermore, the framework offers a runtime which schedules the resulting code variations to their respective PU dynamically.

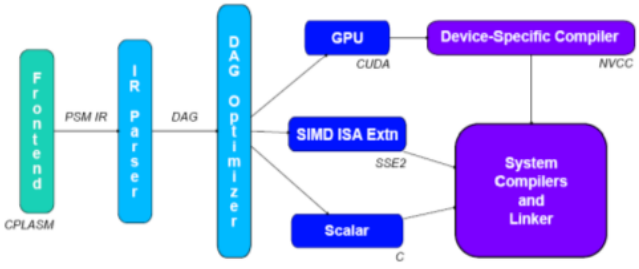


Fig. 10. PLASMA compile-process for CPU, CPU-SIMD and GPU

3.2.1 Performance

The authors compare the results achieved by the PLASMA framework to platform-specific BLAS-libraries. These are ATLAS [9] on the CPU-side and CUBLAS [10] on the GPU-side.

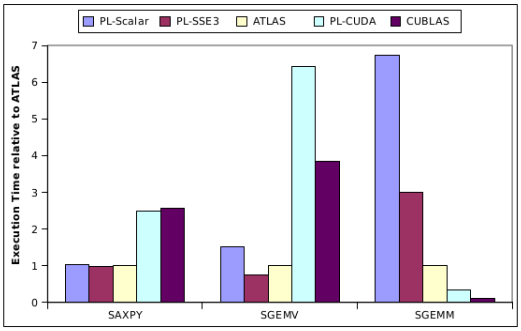


Fig. 11. Plasma Performance [3]

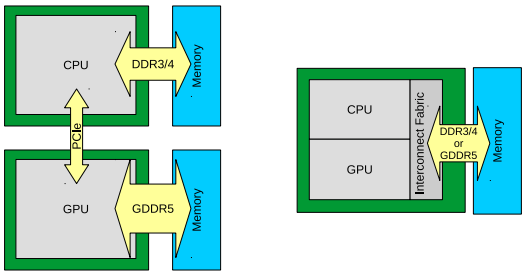
Results for each level of BLAS operation have been published (figure 11). SAXPY represents Vector- (BLAS-1), SGEMV Vector-Matrix- (BLAS-2) and SGEMM Matrix-Matrix- (BLAS-3) Operations. The chart in Fig. 11 shows that the PLASMA framework produces code that is nearly as good as the optimized platform libraries (e.g. SAXPY all, SGEMV PL-SSE3 vs. ATLAS) for specific workloads. Other workloads are much slower (SGEMV PL-CUDA vs. CUBLAS, SGEMM PL-SSE3 vs. ATLAS) with the framework, though. This is a result of special optimizations like cache-blocking which have not been implemented into PLASMA yet.

4 Hardware Solutions

Besides the software solutions described in the previous sections, there are also hardware solutions which address the insufficiencies of CPU/GPU heterogeneous computing systems. Fused CPU/GPU systems address the narrow interconnection of a CPU and a discrete graphics card. Dynamic voltage/frequency scaling is used to lower the energy consumption of CPUs and GPUs to the lowest possible level.

4.1 Fused Heterogeneous Computing Systems

Better know as accelerated processing units (APU), fused heterogeneous computing systems are one approach to address the PCIe bottleneck when it comes to communication between CPU and GPU. These devices combine the CPU and the GPU on one die. This results in significantly shorter communication paths, and therefore better latency and bandwidth. Another advantage of this design is the common, shared address space of CPU and GPU. Explicit, costly data transfers are not necessary. This is illustrated in Fig. 12a and 12b. The width of the yellow arrows indicate the relative bandwidth of the interconnection. As we can see, the narrow PCI-express connection is completely avoided in the case of the fused HCS. The communication between the CPU and the GPU is done via the memory they both share.



(a) Discrete (b) Fused

Fig. 12. Schematic view of discrete and fused CPU/GPU computing systems

Right now, there are several fused HCS solutions from different vendors available.

- AMD Fusion (x86 + OpenCL)
- Intel Sandy Bridge and successors (x86 + OpenCL)
- Nvidia Tegra (ARM + CUDA)

Because the size of a chip can not be scaled unlimitedly, there has to be a trade-off when the CPU, as well as the GPU share the same die. The primary target of these device classes are mainly consumer electronics in the mobile and embedded sector. The overall energy efficiency is therefore inherently good. On the other hand, none of the solutions shown above is nearly as powerful as a high-end CPU in combination with a discrete GPU when we compare the raw performance values. The relative performance of some typical devices is shown in Table 2.

TABLE 2
Relative Performance of fused HCS and dedicated devices

	CPU Perf.	GPU Perf.	fused HCS
AMD A10 7850K	+	++	yes
Intel i7 5775C	++	++	yes
Nvidia Tegra K1	−−	+	yes
Intel Xeon E7-4850	+++	N/A	no
Nvidia K80	N/A	+++	no

The tightly coupled design, however, gives these devices the capability to out-perform discrete solutions in certain application areas. [5], [7] Most of these cases are applications which require a high degree of communication.

4.2 Dynamic Voltage/Frequency Scaling (DVFS)

Modern CPUs and GPUs come with sophisticated power-saving technologies. Clock regions and gating technologies are a way to power-down whole parts of the chip when they are not needed. These techniques are contributing to low energy consumption during idle times of the PU.

DVFS, however, is lowering the power consumption in every operational state of the chip.

$$P = C \cdot V^2 \cdot f|_{f \sim V, C = const} \quad (1)$$

Equation 1 describes the general power consumption of a CMOS chip. The voltage is proportional to the frequency. C is a constant value, caused by the CMOS technology. This

means, if we raise the frequency of a PU, the required power to do so grows cubically. The inversion of this argument means that we can save a lot of energy if we lower the operational frequency of the chip. E.g. if we lower the frequency, f, by 20% the total power consumption, P, drops by 50%.

Modern CPUs and GPUs have plenty of energy states which lower the voltage and frequency to a certain minimum which is required for stable operation. The energy states are adjusted dynamically based on the current load of the PU.

TABLE 3
Energy Consumption of recent high-end GPUs [11]

	Idle	Load
GTX 480	**47W**	235W
GTX 580	**32W**	224W
GTX 680	**15W**	169W
GTX Titan Z (GTX 780Ti)	**13W**	230W
GTX Titan X (GTX 980Ti)	**13W**	240W

Table 3 shows the recent high-end GPUs of Nvidia and their average energy consumption [11] in idle and under load. The idle consumption has been gradually decreasing in the last years. This is due to DVFS and other aggressive energy conservation techniques. Hopefully this trend will continue in the future until we reach a minimal value of as close to zero watts as technically possible.

Therefore, in order to save energy, it is desirable to get the work done as quick as possible, so that the PUs can return to their low-energy states.

5 Conclusion

We have seen different approaches how to run an algorithm cooperatively on a CPU/GPU heterogeneous computing system. CPUs and GPUs are very different in architecture, as well as in the programming model. Intelligent work division is a key factor to gain the maximum benefits from both of them.

Using the low-level languages to program CPUs and GPUs cooperatively is a tedious, error prone process. Frameworks can be used to ease the coding process and raise productivity when it comes to the programming of heterogeneous systems. The benefit of this additional programming effort, however, is in general a better performance.

Discrete CPU/GPU systems offer high peak performance, but suffer from severe bottlenecks. The PCI-Express interface used for the communication between both is comparatively slow and latency dominated. Fused systems which combine CPU and GPU on the same piece of silicon address this issue but have their own down-sides. They are far away from discrete solutions in terms of raw performance. In special applications, however, fused HCS can outperform discrete systems.

Sophisticated energy saving techniques, like DVFS, are also important for the energy-efficiency. They ensure the lowest energy consumption at every operational state of the system.

References

[1] Sparsh Mittal, Jeffrey S. Vetter, *A Survey of CPU-GPU Heterogeneous Computing Techniques*, 2015
[2] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee, *SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters*, 2012
[3] Sreepathi Pai, Ramaswamy Govindarajan, and Matthew Jacob Thazhuthaveetil, *PLASMA: Portable programming for SIMD heterogeneous accelerators*
[4] I. Foster, PCAM, *http://www.mcs.anl.gov/~itf/dbpp/text/node15.html*
[5] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike OConnor, and Tor M. Aamodt, *Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems*, 2012
[6] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun, *An optimized large-scale hy- brid DGEMM design for CPUs and ATI GPUs*, 2012
[7] Mayank Daga, Ashwin M. Aji, and Wu-chun Feng *On the efficiency of a fused CPU+GPU processor (or APU) for parallel computing*, 2011
[8] NASA Advanced Supercomputing Division. NAS Parallel Benchmark, *https://www.nas.nasa.gov/publications/npb.html*
[9] Automatically Tuned Linear Algebra Software (ATLAS), *http://www.netlib.org/atlas/*
[10] NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS), *https://developer.nvidia.com/cublas*
[11] 3DCenter.org, Stromverbrauch aktueller und vergangener Grafikkarten, *http://www.3dcenter.org/artikel/stromverbrauch-aktueller-und-vergangener-grafikkarten*