Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

Energy Efficiency for Heterogeneous Computing

Julius Wagt

June 15, 2020

Supervisor: Ana-Lucia Varbanescu

Abstract

High energy consumption leads to serious environmental concerns, and reducing it is an important technological challenge. To this end, different architectures, like CPUs and GPUs, have different energy consumption characteristics.

Moreover, CPU and GPU architectures have been developed with different kinds of applications in mind, i.e., they have been optimized to perform different tasks. In turn, this means they will also have different performance characteristics when executing a similar workload.

These differences between CPUs and GPUs make heterogeneous platforms and applications (i.e., platforms and applications that use CPUs and GPUs together) challenging to optimize for performance and energy efficiency.

In this work, we focus on analyzing the execution time and energy consumption of a set of heterogeneous applications, implemented with the help of OpenMP and CUDA. The goal of this thesis is to provide an answer to the following question: **Is there tension between performance and energy efficiency in heterogeneous computing?** To answer this question, we propose an empirical approach, where we select four representative workloads as benchmarks, and compare both their performance and the energy consumption on two different heterogeneous machines. We further investigate what the right tools are for measuring energy efficiency.

Our results indicate that there is performance to be gained through the use of CPU+GPU computing. Although we observed that, for benchmarks with low arithmetic intensity that the speedup of their heterogeneous versions is negligible compared to the homogeneous implementation. We have also demonstrated that better energy efficiency comes paired together with better execution times for all our benchmarks. Finally, we have shown that the tested GPUs provide better energy efficiency compared to their respective CPU counterparts.

From our work we conclude that better energy efficiency comes paired with better performance, and the main cause behind this is limited idling (i.e., energy wasted while the processing unit it unoccupied, waiting for the other unit to complete its share). Moreover, the use of heterogeneous computing is limited as the gained speedup is sometimes questionable. However when speedup is achieved, we notice an increase in energy efficiency, too.

Contents

CHAPTER 1

Introduction

1.1 Context

Heterogeneous computing is a generic name for multiple types of processing units working cooperatively on the same workload. A good example of a heterogeneous platform is a CPU+GPU machine, where both units are used to execute the same application. Such architectures are very popular nowadays, because Graphic Processing Units (GPUs) have become ubiquitous as general purpose computing accelerators, due to their high performance compared to regular Central Processing Units (CPUs).

CPU and GPU architectures have been developed with different kinds of applications in mind. This means they have been optimized to perform different tasks. When combining these processing units, it is often clear what the benefit of heterogeneous computing for performance is, because dividing the workload often leads to a shorter execution time. But it is not always entirely clear what these benefits hold in terms of energy efficiency. Deciding how a computation should be split among devices with different capabilities is crucial to minimize both energy consumption and completion time.

In this project, we aim to investigate and potentially improve the energy efficiency of heterogeneous CPU+GPU computing for a set of applications. Therefore, we build heterogeneous versions of these applications, measure their energy efficiency, and systematically determine what is the best workload mapping in terms of energy efficiency on different hardware platforms. It has already been shown [1] that different ratios of work division between CPUs and GPUs may lead to different performance and energy levels. Our ultimate goal is to determine whether there is tension between performance and energy efficiency, i.e., whether there is a loss in performance when energy efficiency comes first.

1.2 Ethics

The ever-growing concerns related to global warming, coupled with the exponential growth of data centers and supercomputing, means that the energy efficiency of these instances has become one of today's major IT issues. Here we shortly address these issues and hope to offer a solution in terms of heterogeneous computing.

1.2.1 Green computing

It is predicted that ICT could use up to 51% of global electricity in 2030 and it could contribute up to 23% of greenhouse gas emissions [2]. Energy efficiency in data centers and supercomputing installations has become one of the major concerns in the past decade, not only because of the monetary cost but also for environmental sustainability. The electricity consumption of these sectors is constantly increasing and there is an urge to apply optimizations in hardware and software to achieve the best performance per watt.

Green computing focuses on improving/maintaining computing performance while reducing energy consumption and the carbon footprint. Technological advances have significantly helped in this aspect: new generation of processors and accelerators are significantly more energy efficient than their predecessors.

Systems could further employ techniques such as dynamic voltage scaling or dynamic frequency scaling to reduce energy consumption at runtime. Such techniques aim to exploit slack time in order to reduce power consumption without degrading performance. In cases where performance is still more important than energy consumption, like HPC, these techniques are only applied when resources are known to be idle/underutilized.

Finally, in systems where heterogeneous computing is available, using the most energy-efficient combination of processing units for a given task is another way to reduce energy consumption. The work presented in this thesis aims to contribute to this selection.

1.2.2 Green 500

Although supercomputers provide an unparalleled level of computational horsepower for solving challenging problems, such horsepower usually equals enormous energy consumption [3], not only to run the supercomputer but also to cool it. This, in turn, results in extremely large electricity bills. For decades, the supercomputing community has focused on performance, and, occasionally on price per performance, where performance is defined as speed. The TOP500 list [4] of the world's fastest supercomputers calculates the speed metric as floating-point operations per second (FLOPS). However, this focus on performance has let other evaluation metrics go unchecked. That is why, since 2007, Green500 [5] has emerged as a complement to TOP500, where supercomputers are ranked by performance-per-watt. Thus, Green500 ranks the top 500 supercomputers in the world by energy efficiency, ultimately rewarding supercomputers that contribute to more sustainable HPC.

1.3 Research Question and Approach

It may seem obvious that splitting a workload among multiple processing units that run in parallel leads, in most cases, to a lower execution time. For CPU+GPU systems, it is often the case that GPUs take the bigger share of the workload, due to their high computational throughput. In this thesis, we set out to investigate what are the implications of such a partitioning for energy efficiency.

Therefore the question this thesis answers is:

Is there tension between performance and energy efficiency in heterogeneous computing?

To answer our question, we propose an empirical approach, where we select a set of representative workloads as benchmarks, and analyze both their performance and energy consumption for different CPU-GPU workload partitioning.

Tension arises when the configurations for peak performance and peak energy efficiency differ significantly. Thus, we envision the following systematic procedure for every application in our set:

1. Implement a heterogeneous version of the given application, thus enabling the CPU and GPU to work concurrently on the same task.
2. Determine the workload partitioning for peak performance, W_p.
3. Determine the peak energy-efficiency workload partitioning, W_e.
4. Compare the two configurations, W_p and W_e, and determine how different they are.

1.4 Outline

The remainder of this thesis is organized as follows. We provide a brief introduction to parallel programming, heterogeneous systems, and energy consumption in Chapter 2. In Chapter 3, we provide a detailed explanation of our methodology and discuss representative workloads. Our experiments and results are discussed in Chapter 4. In Chapter 5 we discuss relevant work and compare our findings against previous work. Finally, in Chapter 6, we conclude our work by answering the posed research question and highlight potential directions for future work in this field.

CHAPTER 2

Theoretical Background

This chapter starts by introducing the basic concepts needed to understand this work. First, we highlight the relevant aspects of the CPU and GPU architectures and describe what defines a heterogeneous application in Section 2.1. We describe the CUDA and OpenMP programming models in Section 2.2. Finally, in Section 2.3 we give a brief overview of energy consumption and the different ways to measure it.

2.1 The Hardware Platforms

2.1.1 CPUs

A Central Processing Unit (CPU), is a component within a computer that executes the instructions that make up any application. The CPU performs basic arithmetic, logic, controlling, and input/output operations specified by the instructions in the program. An overview of a CPU architecture can be found in Figure 2.1.

Hardware multithreading

Multithreading is the ability of a CPU to execute multiple threads concurrently. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the Translation Lookaside Buffer (TLB). Multithreading is mainly used to increase the utilization of the available resources, if a thread gets for example a lot of cache misses, the other threads can continue to take advantage of the unused computing

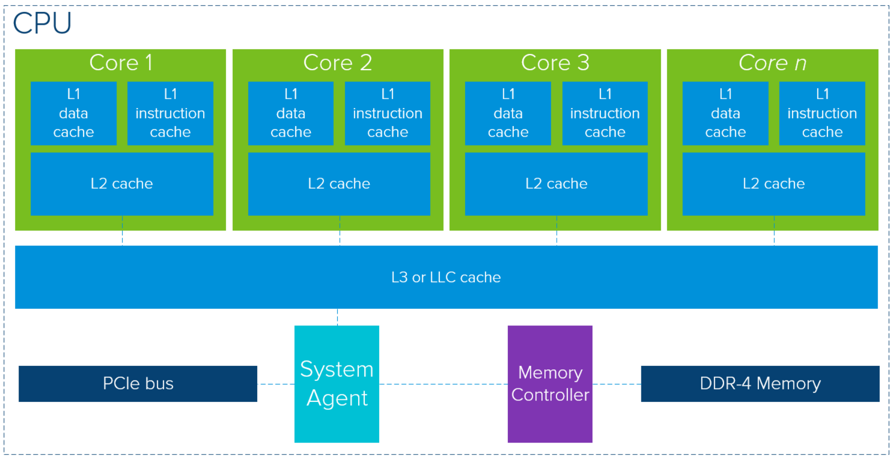


Figure 2.1: Overview of a CPU architecture. Source: Exploring the GPU Architecture [6]

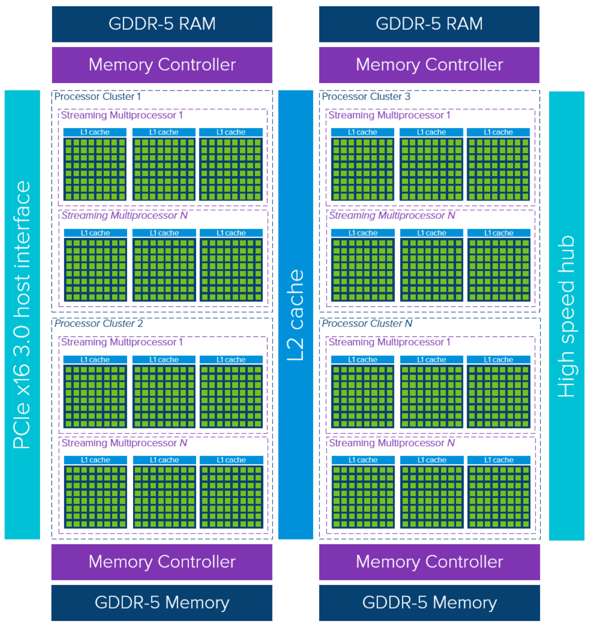


Figure 2.2: Overview of a GPU architecture. Source: Exploring the GPU Architecture [6]

resources. This may lead to faster overall execution, as these resources would have been idle if only a single thread was used.

The most advanced type of multithreading is Simultaneous MultiThreading (SMT), where instructions from more than one thread can be executed at a time. To distinguish the other types of multithreading from SMT, the term temporal multithreading is used to denote when instructions from only one thread can be issued at a time. SMT is the process of a CPU splitting each of its physical cores into virtual cores, which are known as hardware threads. This is done in order to increase performance and allow each core to run multiple instruction streams at once, Intel calls this process hyper-threading. In this new architecture, the CPU must be able to fetch instructions from multiple threads in one cycle, and its register file must be large enough to hold data from multiple threads. The number of concurrent threads is decided by the chip vendors, but two concurrent hardware threads are the most common option supported by regular CPU cores.

Memory model

CPUs have a memory hierarchy, which is divided into multiple levels of memory, the levels closer to the CPU provide faster access time, but they are smaller in size. A basic memory hierarchy contains the main memory at its lowest level, up to three levels of caches, and registers.

A typical CPU package commonly consists of cores, each featuring separate data and instruction layer-1 caches, supported by the layer-2 cache. The layer-3 cache, or last level cache, is shared across all cores in the package. At the highest level of the hierarchy we find the registers. These registers are found directly on the CPU, which means the datapaths between these registers and the Arithmetic Logic Units (ALUs) is extremely short.

Functionality-wise, CPUs fetch data from memory into registers, to further use it for computation. When data is not residing in any cache, the CPU will fetch it from the main memory. As explained, the latency of this fetch grows higher as the level in which the data is found is farther away from the CPU.

2.1.2 GPU's

GPUs are massively multi-threaded many core chips that contain thousands of cores. Historically, GPUs were used for graphic processing. But nowadays with GPUs being so fast and essentially more or less unused processing units, they are used for more general purposes. In the consumer market, GPUs are still mostly bought to accelerate gaming graphics. General Purpose GPUs (GPGPUs) are the choice of hardware to accelerate computational workloads in High Performance Computing (HPC). To facilitate the use of GPGPUs there are two popular frameworks, NVIDIA's CUDA and an open source implementation OpenCL by the Kronos group, both provide a framework to run code on a GPU. For the remainder of this work, we describe heterogeneous systems following the terminology used by CUDA: the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches the *kernels*, which are the functions offloaded for execution on the device.

The reason behind the difference in capability between the CPU and the GPU is that the GPU is specialized for compute intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. A CPU is optimized to be as quick as possible to finish a task at a as low as possible latency, while keeping the ability to quickly switch between operations. This difference makes the CPU and GPU suited for different kinds of problems.

An overview of a GPU architecture can be found in Figure 2.2. A single GPU device consists of multiple Processor Clusters (PC) that contain multiple Streaming Multiprocessors (SM). Typically, one SM uses a dedicated layer-1 cache and a shared layer-2 cache before pulling data from global memory. Compared to a CPU, a GPU works with fewer, and relatively small, memory cache layers. The GPU architecture is tolerant of memory latency, because a GPU has more transistors dedicated to computation meaning it cares less how long it takes to retrieve the data from memory.

The programming model most commonly used when programming a GPU is based on the stream programming model. In the stream programming model, input to and output from a computation comes in the form of streams. A stream is a collection of homogeneous data elements on which some operation, called a kernel, is to be performed, and the operation on one element is independent of the other elements in the stream.

Because of the different programming models between the GPU and the CPU, there are different ways to define execution time. Generally, on CPUs, execution time does not include any operations other than the computation itself, as it is assumed that the data is available and ready to use. For GPUs this is different, first data has to be transferred from the host to the device, and the eventual result has to be transferred back from the device to the host. We call these operations: host-to-device (H2D) and device-to-host (D2H) times respectively. The H2D and D2H times are a form of overhead caused by the use of a GPU. This overhead can be significant if the input data for the GPU is significantly large enough. If a kernel runs faster on the GPU than on the CPU, the H2D and D2H overhead can still make the overall execution time on the GPU slower.

2.1.3 Heterogeneous architectures

The most common heterogeneous systems today usually combine a CPU, with multiple cores, together with a GPU. GPUs and CPUs have been designed for very different kinds of computations: while CPUs are optimized for low latency, GPUs are designed for high throughput. For these reasons, GPUs are made to execute in parallel computational intensive programs. GPUs provide hardware parallelism exploitation, but with a less complex control flow management, like for example branch prediction. While CPUs are composed of few cores, with hardware support only for few execution flows, Therefore, the advantage of heterogeneous systems is that tasks can be offloaded to different Processing Units (PUs) with different capabilities. Such *workload partitioning* leads to performance gain, usually at the cost of an increased complexity of the program itself.

2.2 Programming Models

2.2.1 OpenMP

OpenMP [7] is a set of compiler directives or pragmas that allow programmers to annotate sequential C, C++ or Fortran code. These directives specify the actions to be taken by the compiler and run-time system in order to execute the program in parallel. The directives are designed so that even if the compiler does not support them, the program will still yield correct behaviour, but without any parallelism.

Programming Model

OpenMP operates according to a fork-join model, upon parallel execution. Multiple threads perform tasks defined by OpenMP directives. An OpenMP program begins as a single thread, called the initial thread. An initial thread executes sequentially, as if the code encountered is part of an implicit task region, called an initial task region, that is generated by the implicit parallel region surrounding the whole program.

When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the parallel construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the parallel construct. Only the master thread resumes execution beyond the end of the parallel construct, resuming the task region that was suspended upon encountering the parallel construct. Any number of parallel constructs can be specified in a single program.

In the case of a for loop. The iterations will run in parallel and join at the end of the for loop. This works best for independent iterations, but it is possible to have different forms of synchronization. Reduction operators are built in and programmers are able to define atomic operations and critical sections within the parallel for loop construct. The reduction clause is a mix between the private, shared, and atomic clauses. It allows to accumulate a shared variable without the atomic clause, but the type of accumulation must be specified. It will often produce faster executing code than by using the atomic clause. At the beginning of the parallel block, a private copy is made of the variable and preinitialized to a certain value. At the end of the parallel block, the private copy is atomically merged into the shared variable using the defined operator. The syntax of the clause is: reduction(operator:list).

OpenMP parallelism granularity can be controlled manually by adjusting the number of OpenMP threads in combination with a scheduling type, such as static or dynamic, which insures a coarse-grained parallelism. OpenMP supports both task and data parallelism.

OpenMP 4.0 added explicit SIMD parallelism. SIMD means that multiple calculations will be performed simultaneously by the processor, using special instructions that perform the same calculation to multiple values at once. This is often more efficient than regular instructions that operate on single data values. The for and simd constructs can be combined, to divide the execution of a loop into multiple threads, and then execute those loop slices in parallel using SIMD.

2.2.2 CUDA

Compute Unified Device Architecture (CUDA) is a term used to describe both the parallel computing platform and a set of APIs created by NVIDIA to exploit the computational power of General Purpose Graphics Processing Units (GPGPUs). Using CUDA and the provided proprietary compiler, nvcc, the programmer is allowed to specify the way a thread executes by just using plain C, C++ but also Fortran, together with some proprietary functions for synchronization and atomic operations. Computations on the GPU are expressed in terms of kernels. Threads instantiation, scheduling and termination is entirely managed by the underlying system. Executing code on the GPU usually follows a three step procedure. First data is

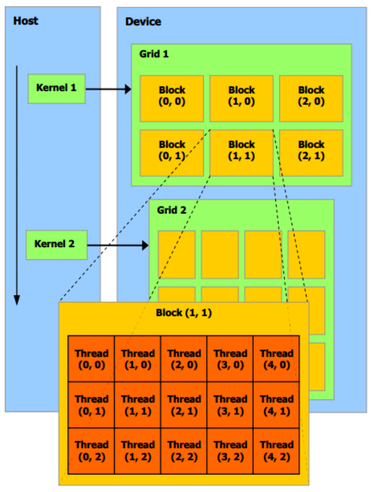


Figure 2.3: Host issues kernel invocations to the device. Each kernel is executed as a batch of threads organized as a gird of thread blocks. Source: the CUDA Programming Guide [8]

transferred from the host memory to device memory, using cudaMemcpy. Then a kernel call triggers the computation on the GPU which operates on the data which was transferred from the host. Lastly the result is transferred back from the device to global memory on the host using again cudaMemcpy.

Programming Model

The CUDA programming model organizes threads into a three-level hierarchy. At the highest level of the hierarchy is the grid. A grid is a 2D array of thread blocks, and thread blocks are in turn 3D arrays of threads. The size of the grid and the thread-blocks are determined by the programmer. Each thread-block in a grid has it's own unique identifier and each thread has a unique identifier within a block. Using a combination of block-id and thread-id, it is possible to distinguish each individual thread running on the entire device. An example of a grid of thread blocks executing a kernel is illustrated in Figure 2.3.

A thread block is composed of warps. A warp is a set of 32 threads within a thread block such that all the threads in a warp execute the same instruction at the same time. In other words, threads cannot diverge. Unlike grids and blocks, warps are an implementation detail that is not directly accessible by the programmer. Code that involves branching can complicate the execution flow of a warp, this is because of the requirement that threads in a warp can not diverge. CUDA fixes this problem by instructing the warp to execute both branches of a conditional statement, this means the code is not executed in parallel, but in serial. This serialization can result in a significant performance loss. When branching occurs a lot in the program, the GPU will be inefficient.

Threads can cooperate with other threads in the same block using global synchronization barriers and shared memory. Shared memory is similar to cache memory in CPUs, except it can be managed explicitly. Since device memory is of much higher latency and lower bandwidth than on-chip memory, device memory accesses should be minimized and the use of shared memory should be encouraged. So a programmer would then load the data from device memory to shared memory, for each thread in a block. Then synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were written by different threads. Then process the data that has been loaded into shared memory and finally write the results back to device memory.

The independence of blocks and kernels allows CUDA to be more scalable and more flexible. It allows for thread level task parallelism, as different threads can execute different tasks and also block and grid level parallelism since different blocks or grids can also execute different tasks. It also allows for data parallelism as different threads and blocks process different parts of data in memory.

2.3 Energy Consumption

In physics, power is the rate of doing work, and it is equivalent to an amount of energy consumed per unit time. In SI units, energy is measured in joule, and power is measured in watts, which is equivalent to joule per second. Accurate measurement of energy consumption during an application execution is key to energy minimization techniques at software level.

Muhammad Fahad et al. [9] discussed three popular approaches to measuring energy consumption. The first is *system-level physical measurements using external power meters*, which they consider to be the ground truth. The second approach includes *measurements using on-chip power sensors*. The third and final class are *energy predictive models*.

Besides the difficulties of purchasing, deploying, and using external power meters, the first approach can only provide the measurement at a computer level, and therefore lacks the ability to provide fine-grained measurements. If the application were to execute on several computing devices, then this approach would not be able to provide the energy consumption per device, making it difficult to determine per-device individual contributions.

The second approach is based on on-chip power sensors which are now provided in mainstream processors. The Running Average Power Limit (RAPL) interface is a feature introduced in the Intel Sandy Bridge architecture, and is also available on more recent AMD architectures such as Ryzen and Epyc. RAPL provides two different functionalities. First, it allows energy consumption to be measured at very fine granularity and a high sampling rate. Second, it allows limiting the average power consumption of different components inside the processor, which also limits the thermal output of the processor. A recent paper [10] conducted a comprehensive study to demonstrate how RAPL performs in terms of accuracy, performance, granularity and usability. The authors conclude that RAPL is a useful tool that can provide accurate enough results for the power consumption of a CPU, and that its limitations are in most cases negligible. The NVIDIA System Management Interface (nvidia-smi) is a command line utility, that offers a way to obtain the energy consumption of an NVIDIA GPU from its on-chip power sensors. The built-in sensors queried by nvidia-smi are not highly accurate, an error margin of +/- 5% should be assumed. These queries are also done not as often as the RAPL updates, making nvidia-smi more coarse-grained compared to RAPL.

The RAPL interface uses performance monitoring counters (PMCs) as predictor variables to measure energy consumption. RAPL provides the following power domains for both measuring and limiting energy consumption. These power domains can also be seen in Figure 2.4.

- Package: Package (PKG) domain measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics and also the uncore components (L3 cache, memory controller).
- Power Plane 0: Power Plane 0 (PP0) domain measures the energy consumption of all processor cores on the socket.
- Power Plane 1: Power Plane 1 (PP1) domain measures the energy consumption of processor graphics (GPU) on the socket.
- DRAM: DRAM domain measures the energy consumption of random access memory (RAM) attached to the integrated memory controller.

Performance counters offer a way to count hardware events during code execution on a processor, allowing for an in-depth view on what happens on the processor while running applications. Since this mechanism is implemented directly in hardware there is no overhead involved. Each power domain reports the energy consumption of the domain, allowing the programmer to limit the power consumption of that domain over a specified time window. RAPL also monitors the performance impact of the power limit.

The RAPL energy counters can be accessed through model-specific registers (MSRs). The counters are 32-bit registers that indicate the energy consumed since the processor was booted up.

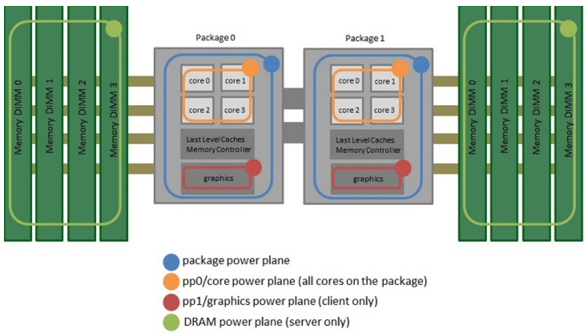


Figure 2.4: Power Planes of a CPU. Source: Energy Consumption Measurement of C/C++ Programs Using Clang Tooling [11]

The counters are updated approximately once every 1 millisecond. The MSRs can be accessed directly on Linux using the MSR driver. For direct MSR access the MSR driver must be enabled and the read access permission must be set for the driver. The MSR driver supports raw access to any MSR including the RAPL energy counters. It is nearly universally supported regardless of the kernel version. Hähnel et al. [12] evaluated whether RAPL can be used to measure short code paths. They showed that RAPL updates do not occur precisely every 1 millisecond but instead they have some jitter. They also compared the RAPL measurements with external measurements with a manually instrumented board and showed that RAPL measurements do correlate nicely with external measurement with a fixed offset.

In summary using on-chip sensors with the help of RAPL is a good option to measure the energy consumption of an application. The RAPL energy calculations are implemented in the hardware and since RAPL is always running, energy consumption can be measured on the same machine without significant overhead. RAPL updates the energy counters approximately once every 1ms. This frequency is much higher compared to external power meters which typically measure power only once a second. RAPL starts running as soon as the processor boots up, which means there is no need to configure it, and no additional equipment is needed, which makes RAPL a very low cost option.

CHAPTER 3

Heterogeneous Applications and Tools

Our approach to assess energy efficiency for heterogeneous computing is based on empirical evaluation. Therefore, we need to select a set of representative applications and tools to be used for this assessment. In Section 3.1, we describe in detail the applications we use. Representative workloads will be discussed in Section 3.2. Finally, in Section 3.3, we describe the tools and procedures we use to monitor and measure the energy consumption.

3.1 SHOC

To design and validate our methodology, we focus on a set applications from the Scalable HeterOgeneous Computing (SHOC) [13] benchmark suite. The SHOC benchmark suite offers a wide variety of benchmarks. In addition to a serial version of each benchmark, SHOC also provides an embarrassingly parallel version (which executes on different PUs or nodes of a cluster, but have no communication between PUs or nodes), and a true parallel version (which measures multiple nodes, with single or multiple PUs per node, and also involves communication). This means that SHOC benchmarks scale from a single PU to a large cluster, something other benchmarks do not provide. The SHOC benchmark suite is divided into two primary categories: stress tests and performance tests. The stress tests use computationally demanding kernels to identify devices with bad memory, insufficient cooling, or other component problems. The performance tests are further subdivided according to their complexity. The levels are:

- Stability Tests
- Performance Tests:
  - Level 0: Very low level device characteristics such as bandwidth across the bus connecting the GPU to the host or peak floating point operations per second.
  - Level 1: Device performance for low-level operations such as vector dot products and sorting operations.
  - Level 2: Device performance for real application kernels.

From the many applications that SHOC offers, we selected a subset of three applications from the level 1 performance tests. Level 1 benchmarks contain basic parallel algorithms, which represent common tasks in parallel processing and are found in a significant portion of real applications. We believe with these low-level tests we can best determine whether there is tension between performance and energy efficiency. We briefly describe each application in the following sections. These application are in increasing order of their computational (arithmetic) intensity, according to the Roofline model [14].

3.1.1 Reduction

In computer science, the reduction operator is a type of operator that is commonly used in parallel programming to reduce the elements of an array into a single result; for example, it can be a summation algorithm that computes the sum of large arrays. The reduction operation is presented in Listing 3.1. Many efforts have been made in order to optimize this problem for the GPU [15] [16]. Because reduction is a very common and important primitive, NVIDIA has come up with their own implementation for this problem. Their implementation uses a tree-based approach within each thread block where each thread block decomposes into multiple kernels in order to keep the device as busy as possible. Further speed-up is achieved by writing more memory efficient code and exploiting space-time tradeoffs like loop unrolling.

Listing 3.1: Reduction operation

```
    sum = 0;
    for (i = 0; i < array_size; i++) {
        sum += data[i];
    }
    return sum;
```

3.1.2 Stencil

Stencil codes perform operations to update all elements of a given array using neighbouring array elements in a fixed pattern, called *stencil.* The radius of the stencil is often called the halo, in the case that this halo exceeds the boundary of the array then the values for which we operate the stencil are usually left unchanged. The stencil operation is presented in Listing 3.2. Since the stencil is the same for each element, the pattern of data accesses is repeated, which makes it excellent for parallel computing. There are numerous types of popular stencils. In our case, we will be using a 5-point stencil in both our 1D and 2D applications.

Listing 3.2: Stencil operation

```
    for (y = 0; y < array_size; y++) {
        for (offset = -HALO; offset < HALO; offset++) {
            if (y + offset < 0 || y + offset >= array_size) {
                continue;
            }
            out[y] += in[y + offset];
        }
    }
```

3.1.3 GEMM

GEMM is a general matrix-matrix multiplication of the form:

$$C = \alpha AB + \beta C$$

where A, B are matrices in column-major format and α, β are scalars. A is a n x m matrix, which means that it has n rows and m columns, in this case B is a m x k matrix. The result of the multiplication $A \cdot B$ is a n x k matrix, which should also be the form of matrix C. Figure 3.1 illustrates the idea of a matrix-matrix multiplication and Listing 3.1 presents the GEMM operation. It should be clear why this type of operation is a good example for parallel computation. Each element in C will have to be calculated and independent from the other cells.

Listing 3.3: GEMM operation

```
    for (i = 0; i < side_size; i++) {
        for (k = 0; k < side_size; k++) {
            A_PART = A[i * side_size + k];
            for (j = 0; j < side_size; j++) {
                C[i * side_size + j] += A_PART * B[k * side_size + j];
            }
        }
    }
```

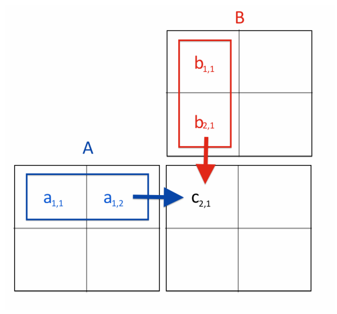


Figure 3.1: Matrix multiplication. Source: Matrix-Matrix Multiplication on the GPU with Nvidia CUDA [17].

3.2 Workload Partitioning

CPU and GPU architectures have been developed with different kinds of application in mind; this means that they are optimized for different tasks and will have, depending on the computation, different performance and energy footprints. Deciding how a computation should be split among devices with different capabilities is crucial to minimize completion time.

3.2.1 Design

In a parallel, heterogeneous application, CPUs should be preferred for executing the tasks with unpredictable or complex control flows, or any sequential portions of the program. On the other hand, significant improvements can be achieved by exploiting the GPU on data parallel, compute intensive tasks. Therefore GPU+CPU heterogeneous architectures offer some unique opportunities in terms of energy conservation and performance. Since GPUs have energy efficiency advantage over CPUs for parallel and computation-intensive applications, an intuitive solution seems to be assigning all those workloads to the GPU for energy efficiency. However the GPU taking all the workloads is not necessarily the most efficient workload division. The main reason is that if the GPU takes all the workloads while the CPU is totally idling, the execution time of the entire system can be longer than that in the case when the CPU does a fair portion of work. Since energy is the product of time and power, a more energy efficient solution is to split and distribute the workload to both the GPU and CPU, such that both sides can finish approximately at the same time. However, because CPUs and GPUs differ considerably in their processing and memory capabilities, meaning this perfect workload division is different for every application.

3.2.2 Implementation

The implementation of each of our heterogeneous programs includes a combination of CUDA and OpenMP. In our case, we are merely offloading computation to the GPU in order to achieve a performance increase. *Offloading* is the process of moving the data from the CPU onto the GPU, performing computation on the GPU, and moving the results back to the CPU. To determine which portion of the input is to be executed by the GPU, we use static partitioning instead of dynamic partitioning, because we do not want the extra scheduling overhead at run-time.

To actually divide the workload we employ a block distribution. This means the data is chopped into blocks, in this case one for the CPU and one for the GPU, which then start working on their respective chunks roughly at the same time. Other distributions could be a cyclic distribution, where small parts of the data are distributed in ”rounds” to each PU. A cyclic decomposition is however terrible for data access, especially for the stencil benchmark, as the halos have to be included for each part of the input data. The block decomposition is much better for all data access patterns present in our applications.

3.2.3 Measuring performance

Listing 3.4 illustrates how we calculate the execution times, of a heterogeneous application. The execution time of the application can be determined accurately using the Chrono [18] high_resolution_clock from the C++ utility library, which measures up to nanoseconds. The GPU kernel is non-blocking, meaning the CPU can execute its portion of the input alongside the GPU. The execution times in our experiments is the summation of the maximum between the GPU kernel and the CPU part, together with the time it took to write the data from the host to the device and back.

Listing 3.4: Timing heterogeneous applications

```
start_H2D_timer;
cudaMemcpy(copy_H2D);
stop_H2D_timer;

start_kernel_timer;
for (int i = 0; i < iterations; i++) {
  kernel<<<>>>();
  cpu_code();
}
cudaDeviceSynchronize();
stop_kernel_timer;

start_D2H_timer;
cudaMemcpy(copy_D2H);
stop_D2H_timer;
```

3.3 Monitoring Energy Consumption

We require a set of tools in order to monitor energy consumption for both the CPU and GPU.

3.3.1 LIKWID

LIKWID [19] stands for "Like I Knew What I'm Doing". It is an easy to use, yet powerful command line performance tool suite for the GNU/Linux operating system and is suitable for Intel and AMD processor architectures. Performance API (PAPI) [20] is another popular and well known framework to measure hardware performance counter data. In contrast to LIKWID, PAPI relies on other software to implement the architecture-specific parts and concentrates on providing a portable interface to performance metrics on various platforms and architectures. The most important difference is that LIKWID's main focus is in providing a collection of command line tools for the end user, while PAPI's main focus is to be used as a library by other tools. In this work, we use a subset of the LIKWID tools.

To record performance data, we use the likwid-perfctr tool, which collects performance counter metrics over the complete run-time of an application. It does this through the Linux MSR module, an interface to access model specific registers from user space, which allows us to read out hardware performance counters. As mentioned before (in Chapter 2), the MSR driver supports raw access to any MSR including the RAPL energy counters.

Implementation-wise, likwid-perfctr is a command line tool that can be used as a wrapper to an application. It allows simultaneous measurements on multiple cores. Events are specified on the command line, and the number of events to count concurrently is limited by the number of performance counters on the CPU. Figure A.1 shows an example of the output from the likwid-perfctr tool.

3.3.2 NVIDIA System Management Interface

The NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML) [21]. This utility offers a way to obtain the energy consumption of an NVIDIA GPU from its on-chip power sensors. The NVML manual reports that the measurements done by nvidia-smi for the power draw of the GPU are accurate to within +/- 5 %.

Our command to query data from the GPU looks as follows:

```
nvidia-smi --query-gpu=timestamp, fan.speed, utilization.gpu, utilization.
    memory, temperature.gpu, power.draw --format=csv,noheader -lms 500
```

The command specifies we want to query a set of properties. These properties follow the selective `--query-gpu=` option and are separated by a comma. After these properties we specify how we would save this data to a file, namely in csv form without the column headers. At the end we specify we want to query this command every 500 milliseconds to the GPU, in order to gain more accurate data. The properties we focus ourselves on are the power draw of the GPU, the other properties guarantee our results are consistent throughout the experiment. Figure A.1 shows example output from the nvidia-smi tool.

3.3.3 Measuring Energy Consumption

For our work, we measure the total energy consumption instead of the dynamic energy consumption. A reason for looking at the dynamic energy consumption only, is because static energy consumption is a constant based on the configuration, therefore it cannot be optimized and it does not depend on the application running on the configuration. However, as we are not focusing on any improvements to either the execution times or the energy consumption of the application, total energy consumption seems to be the more relevant metric.

Using the measurements provided by the on-chip power meters, we determine the total energy consumption during application execution by using equation 3.1, where E_{total} is the total energy consumption of the platform during the execution of an application, P_{static} is the static power consumption of the platform during the execution of an application (i.e., the power consumption of the platform when it is idle), and, finally, T is the execution time of the application.

$$E_{total} = E_{dynamic} + (P_{static} * T) \tag{3.1}$$

The accuracy of obtaining the total energy consumption E_{total} is equal to the accuracy provided in the specification of the on-chip power meter. While the accuracy of GPU on-chip sensors is reported in the NVML manual, the accuracy of RAPL for CPUs is not known. For both tools, there is no information about how the power reading is determined, meaning there is no clear way how to determine the accuracy of such a reading. Therefore, the accuracy of on-chip sensors needs to be thoroughly validated were they to be used for the optimization of applications in terms of dynamic energy.

In order to measure the energy efficiency of our machines we propose two metrics: Energy Delay Product (EDP) and FLOPS per watt [22]. EDP combines both energy consumption and performance in one metric, simply by multiplying them. This product between the execution time and energy consumption illustrates the relationship between the two, since both factors are equally stressed. Moreover, lower EDP translates into better energy efficiency of the code under study. FLOPS per watt measures the amount of performance gained for every watt of power drawn, rather than weighting a component of energy. As expected, higher FLOPs per watt translates into better energy efficiency. For the purpose of this work, we will not calculate or measure a very accurate number of FLOPs. Instead, we will use a theoretical estimate based on a high-level analysis of the application source code. This theoretical estimate is based on the computational (arithmetic) intensity, according to the Roofline model [14].

CHAPTER 4

Energy Efficiency Analysis

In this chapter we present an empirical analysis of our three benchmarks. Our experiments are focused on measuring the performance and energy consumption of three heterogeneous applications on two different configurations.

4.1 Experimental Setup

4.1.1 Hardware and software

We run all implementations using heterogeneous systems, and employing both the CPU and the GPU in parallel. All the experiments in this thesis have been conducted on two heterogeneous machines: PM (personal machine) and DM (DAS machine). The PM contains a octa-core AMD Ryzen 7 1800X CPU, and the GPU is a NVIDIA GeForce GTX 1080 Ti. The DM is provided to us by the DAS5 [23], containing a dual octa-core Intel Xeon E5-2630-v3 CPU and a NVIDIA GTX TitanX GPU. More details about the used hardware can be found in Table 4.1. We mentioned in Section 3.3.1 RAPL monitors performance counters found on the CPU chip, however the names of these performance counters differ per architecture. The AMD Zen microarchitecture provides 2 energy counters for CPU core and package energy, the names for which are: EVENT_RAPL_CORE_ENERGY and EVENT_RAPL_PKG_ENERGY. The CPU core counter returns one value per CPU core, the package counter returns one value per CPU socket. The Haswell microarchitecture provides 4 energy counters for all the power plane domains of the CPU discussed in Section 2.3. The names of these counters are: EVENT_PWR_PKG_ENERGY, EVENT_PWR_PP0_ENERGY, EVENT_PWR_PP1_ENERGY and EVENT_PWR_DRAM_ENERGY.

The GPUs are programmed using CUDA version 10.1 with Linux x86_64 Driver Version 435.21. We compile all host code with the g++ compiler, version 7.5.0, using the -Ofast flag; the used OpenMP version is 4.5.

Type	GPU	CPU
Name	NVIDIA GeForce GTX 1080 Ti	AMD Ryzen 7 1800X
(Micro)Architecture	Pascal	Zen
Clock Frequency	1481 MHz	3600 MHz
Cores per Socket	3584	8
Idle Power	13 watt	13 watt
Name	NVIDIA GTX Titan X	Intel Xeon E5-2630 v3
(Micro)Architecture	Maxwell	Haswell
Clock Frequency	1000 MHz	2400 MHz
Cores per Socket	3072	8
Static Power	9 watt	12 watt

Table 4.1: The hardware of the PM and DM.

Name	Input size	Iterations
Reduction	150000	1000000
Stencil 1D	150000	1000000
Stencil 2D	2000 x 2000	5000
GEMM	2000 x 2000	2500

Table 4.2: The selected parameters for each benchmark.

The full command we compile our code with is:

```
nvcc -Xcompiler -Ofast -Xcompiler -fopenmp -lcuda -lgomp -o program program.cu
```

It is the purpose of nvcc, the CUDA compiler, to hide the intricate details of CUDA compilation from developers. All non-CUDA compilation steps are forwarded to a C++ host compiler that is supported by nvcc. Xcompiler allows for passing specific options directly to the internal compilation tools that nvcc encapsulates, without burdening nvcc with too detailed knowledge on these tools. The parameter flag -Ofast enables all optimizations for compiling. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The -lcuda and -lgomp link to their respective libraries. The -fopenmp flag is needed to activate the OpenMP extensions for C/C++, thus enabling the use of the OpenMP directive `#pragma omp` in C/C++.

4.1.2 Benchmarks

All of the chosen benchmarks were run with a fixed but different amount of iterations and input sizes. These parameters vary per benchmark because of the computational complexity within the benchmark. Moreover we fixed the execution configuration for the CPU and GPU (i.e. the number of threads and block/grid geometry for the CPU and GPU, respectively). When we changed our configuration we did indeed observe a different optimal workload partitioning, as these parameters influence the performance of both the PUs by having more resources available. We did not however observe a different behaviour for the curve, thus proving our method can be applied for any such configuration. Determining an optimal configuration requires extensive auto-tuning, which is beyond the scope of this thesis.

Table 4.2 shows the parameters used per benchmark. Important is we run a decent amount of iterations to obtain more reliable results. If we run too many iterations we have the problem that the performance increase, for a different workload distribution, is more difficult to observe.

We find an optimal β, which represents the workload partitioning of our input, for each benchmark. The process to obtain the optimal β is as follows: we run the benchmarks with the parameters from Table 4.2 and observe where the execution time is lowest. We consider this the optimal workload partitioning. We do the same in terms of energy consumption, naming the point where the energy consumption is lowest η.

The applications used in this thesis have been described in detail in Chapter 3. For each application, we test its heterogeneous version using 20 different workload partitionings, ranging from 0-50% where 50% means a perfect split of the workload between the CPU-GPU and 0% meaning all work will be run on the GPU. We take steps of 2% between 0% and 30% then we take steps of 5% until 50%, this is to provide fine grained analysis. Preliminary results of our benchmarks have shown that the optimal beta was always between 0 and 50%, which is why we have kept our focus in this range.

During each experiment, alongside the execution times, we also measure the energy consumption during the entire experiment. The total energy is the sum of the average values measured for the CPU and the GPU, over multiple runs. This energy describes the energy from both PUs plus the energy from when they are idle. In Figure 4.1 we can see the power draw of the GPU when it is idling. Results for the CPU can vary as the CPU is never truly idle. Since we observe that the idle energy from the GPU is mostly constant, we see it as a constant increase for the total energy.

We have also measured the wall clock time, but we observed that this time was a constant 1.5 seconds longer than the measurements specifically targeting the compute kernel. This overhead

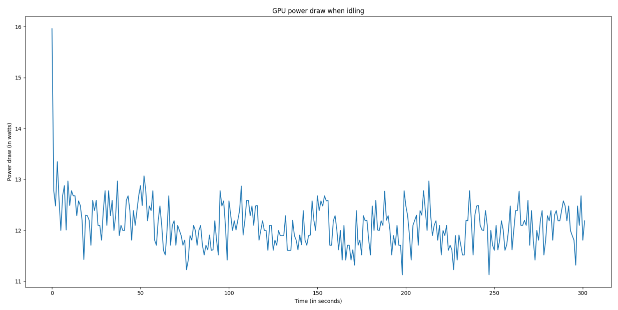


Figure 4.1: Power Draw of the GPU during idling

appears because, in order to get reliable time measurements, likwid-perfctr 3.3.1 must determine the base clock frequency of the CPU. This is done by a measurement loop that takes about 1 second. We must also initialize the input data. Nevertheless, since this is a constant overhead, we omit it from our results.

During most experiments we noticed the GPU was not running at a 100% utility rate. This can have multiple reasons. It seems that the GPU takes a while to run at 100% utility. The reason it takes fairly long for the GPU to reach this steady state is because of the heat capacity of the entire product, which has quite a bit of mass. To "warm up" the GPU, we ran an additional kernel before our experiments, but this did not help. Because the GPU is also heating up under the sustained load of our experiments, the energy consumption might "artificially" increase over the course of our experiments, as more power is needed for the cooling of the GPU. The GPU components draw more power at increased temperature mostly due to an increase in Ohmic resistance. In addition, as the GPU heats up, we would expect the fans on the card to spin faster and therefore require more power, however we did not observe this in our results.

We note that, ultimately, it is not necessary that this steady state power consumption is achieved for our experiments, as long as the conditions under which each benchmark is tested is as equal as is practically achievable.

4.2 App1: Reduction

Reduction is the lowest arithmetic intensity benchmark we have tested: it only performs a single operation on each element of the input array. Therefore, we must run this benchmark with more iterations. We observe in Figure 4.2 the low arithmetic intensity of this benchmark: even with a split workload, we do not really gain a significant speedup. When we let the CPU handle around 16% and 30% of the workload, for the PM and the DM configuration, respectively, we start observing the CPU taking longer than the GPU to finish computing. From that point on, we see a linear increase in the execution time, which is in line with the computational complexity of reduction.

In Figure 4.3, we observe that the GPU has a large power draw for all tested β values, which does not significantly decrease. This happens because we run experiments with different values of β "back-to-back", and the GPU has no time to cool down sufficiently between experiments. As the kernels we test take longer to execute (see GEMM, in Section 4.4), this effect diminishes significantly.

Figure 4.4 shows the energy efficiency per workload partition. For the Energy Delay Product (lower is better) we observe that PM is more energy efficient. This happens because the execution time on the PM is almost half of that of the DM for lower values of β. Both plots also indicate that, for reduction, most/all work should be done on the GPU: the CPU has *no impact for improving the energy efficiency* of this application, for any β.

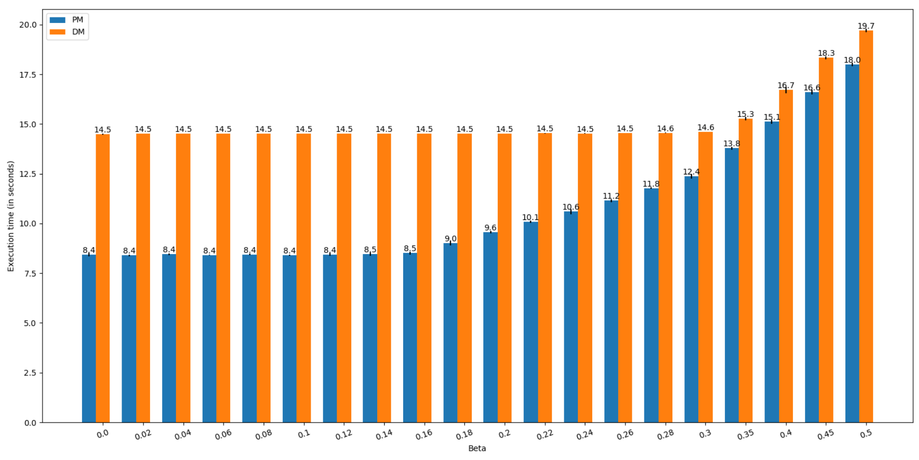


Figure 4.2: Execution times for the reduction experiment.

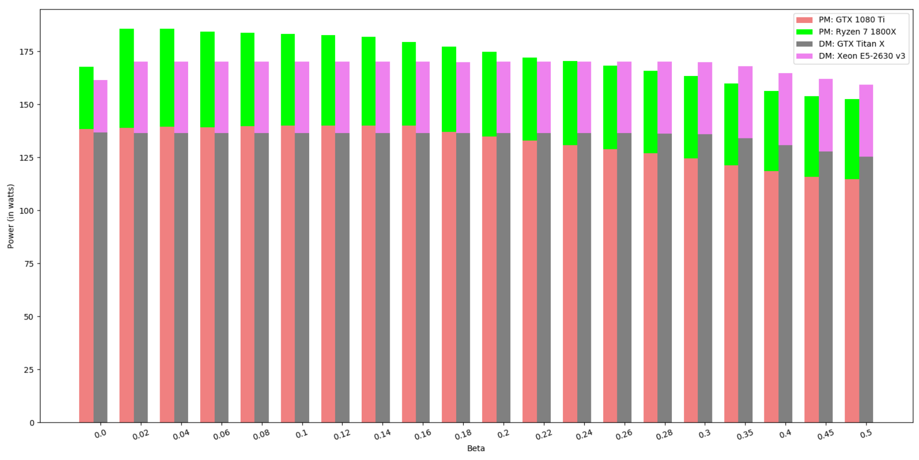
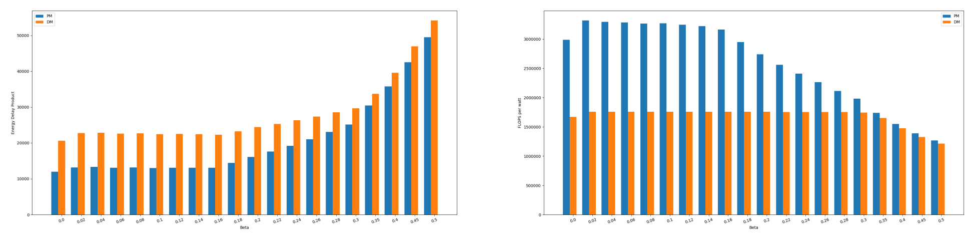


Figure 4.3: Total power draw for the reduction experiment.

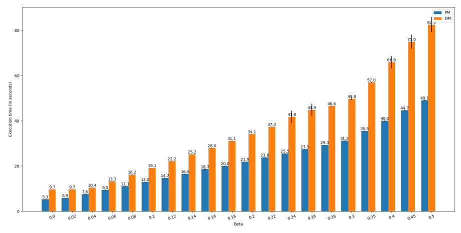


(a) Energy Delay Product (b) FLOPS per watt

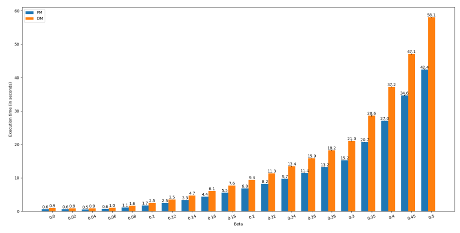
Figure 4.4: Energy efficiency in terms of Energy Delay Product and FLOPS per watt for the reduction experiment.

4.3 App2: Stencil

The 1D and 2D stencil benchmarks have higher arithmetic intensity than reduction, but they are far from matrix multiplication in this respect. We observe no speed-up for the stencil1D benchmark when splitting the workload between the CPU and GPU. For the stencil2D benchmark, we only see a slight speed-up when β is 4%. The slight increase in arithmetic intensity is however visible in that *the CPU's negative impact on performance* is more significant, for both PM and DM, than in the case of reduction. In other words, using the CPU leads to faster performance degradation.



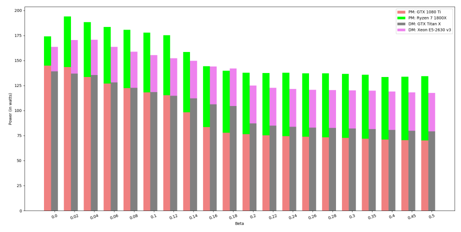
(a) Stencil1D



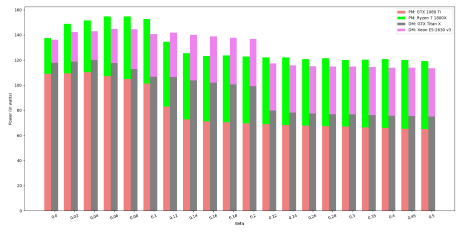
(b) Stencil2D

Figure 4.5: Execution times for the stencil experiment.

In Figure 4.6, we observe that the total average power draw goes down for larger values of β. These are the cases when the GPU would be idle for the most time, therefore lowering the average power draw of the GPU.



(a) Stencil1D

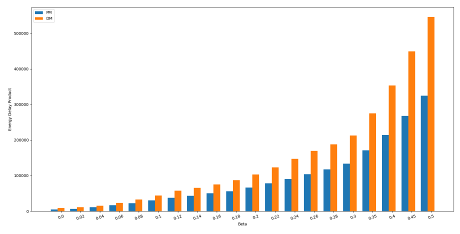


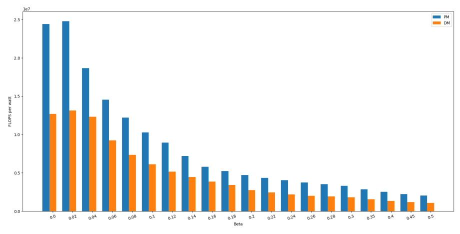
(b) Stencil2D

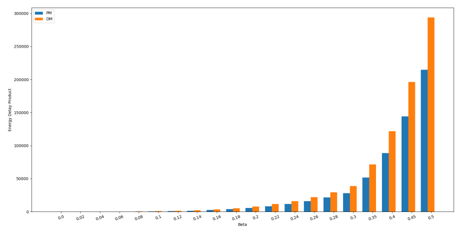
Figure 4.6: Total power draw for the stencil 1D and 2D experiments.

Finally, we analyse energy efficiency, based on the results presented in Figure 4.7. The EDP for the Stencil1D benchmark indicates that the most energy efficient way to execute this application is using only the GPU. The stencil2D plot shows a slower increase in EDP (i.e., a slow decrease of energy efficiency), meaning a heterogeneous version would only have a minor impact in terms of decreasing energy efficiency. We again observe the same trends for both platforms, but notice specific differences in EDP between the PM and DM, with PM outperforming DM.

A similar analysis can be done for our second efficiency metric: FLOPs per watt. We observe the best FLOPs per watt is achieved for the lowest β values, further indicating the GPU should be used for energy efficient computing. For PM, the CPU can bring a minor contribution in improving FLOPS per watt (the optimal value is obtained for $beta = 0,04$), for both platforms. Also here, the PM and DM results are similar in trend, but the significantly lower execution times for PM make this configuration more suited in terms of energy efficiency.


(a) Energy Delay Product Stencil1D


(b) FLOPS per watt Stencil1D


(c) Energy Delay Product Stencil2D

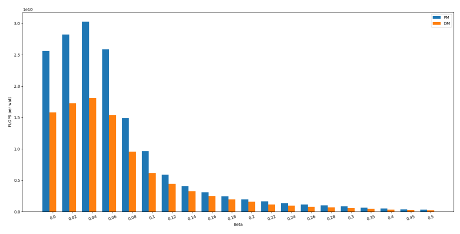

(d) FLOPS per watt Stencil2D

Figure 4.7: Energy efficiency in terms of Energy Delay Product and FLOPS per watt for the stencil experiment.

4.4 App3: GEMM

Matrix multiplication (i.e., GEMM) is the most computation-intensive application we have analysed. The execution times for GEMM are shown in Figure 4.8. We can observe there is a significant variation of the GEMM execution time as we vary the amount of work we push to the CPU. Based on our empirical analysis, the optimal workload distribution between the CPU and the GPU happens when we let the CPU execute around 26% of the input data. Interestingly enough, this point is the same for both PM and DM.

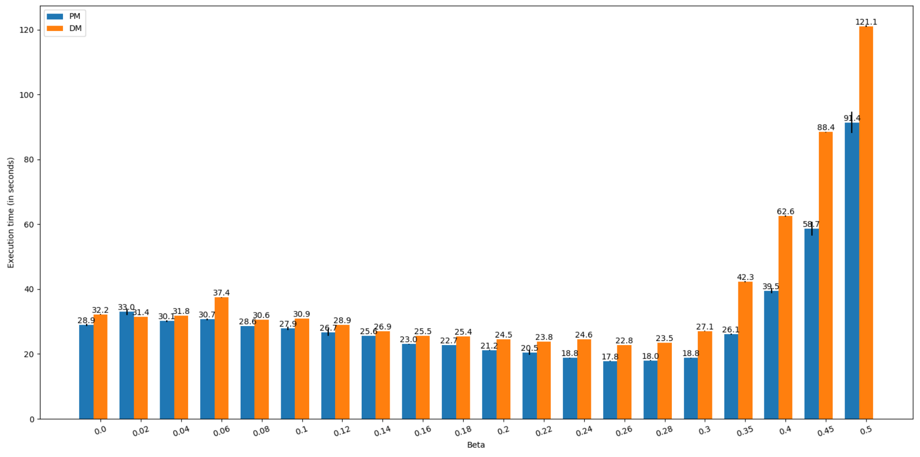


Figure 4.8: Execution times for the GEMM experiment.

We further see that, for larger values of β, the CPU will take increasingly longer to compute than the GPU, thus dominating the compute time of the heterogeneous application. For these larger β values, the execution time grows exponentially, which corresponds with the computational complexity of the GEMM operation.

In Figure 4.9 we observe the total power draw for both hardware platforms, for different values of β. We observe that as β increases, the power draw gets considerably lower. This is because the GPU will start idling for long periods of time (beyond the optimal workload distribution,

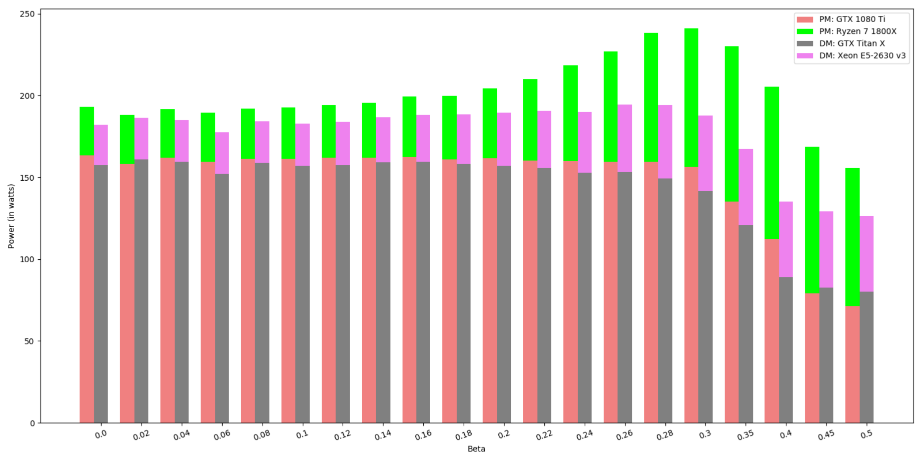
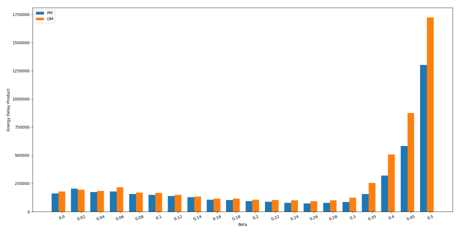
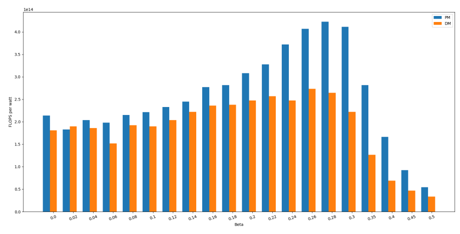


Figure 4.9: Total power draw for the GEMM experiment.



(a) Energy Delay Product



(b) FLOPS per watt

Figure 4.10: Energy efficiency in terms of Energy Delay Product and FLOPS per watt for the GEMM experiment.

the CPU takes longer to execute, and the GPU gets to idle longer). The average power draw is the highest around the optimal workload partition, since that is when both PUs are idling the least amount of time.

Figure 4.10 shows two plots illustrating energy efficiency using our two metrics, Energy Delay Product and FLOPS per watt. Looking at the Energy Delay Product (EDP) results, we observe the most energy efficient configuration is the one with the lowest execution time; moreover, we also note that the GPU is significantly more energy efficient compared to the CPU, to the point where not using the CPU would only lead to a small decrease in energy efficiency, while not using the GPU would lead to a much more significant loss. In the FLOPS per watt plot, we observe that highest efficiency is still obtained for the optimal workload partitioning. However, in this case, the loss in efficiency when not using the CPU seems more significant.

CPU threading

We noticed that the GPU idling when the CPU takes more work is a big part why the energy efficiency leaned more to the GPU side. Therefore, we devised an additional experiment to test what the best CPU configuration is when combining a CPU with GPU in heterogeneous computing.

We only run this experiment for the GEMM benchmark, because the impact of the CPU is the most significant in this application; moreover, without loss of generality (as the trends we have seen so far are similar on DM and PM), we only use DM for this analysis.

To determine the best CPU configuration, we keep the same configuration as we did in the other GEMM experiments, but vary the number of active CPU threads. Specifically, we use 4, 8, 16, and 32 threads; moreover, for we tested two configurations for 16 threads: ”16-1”, where we map all thredas on the same socket (thus, using hyper-threading), and 16-2, where we split

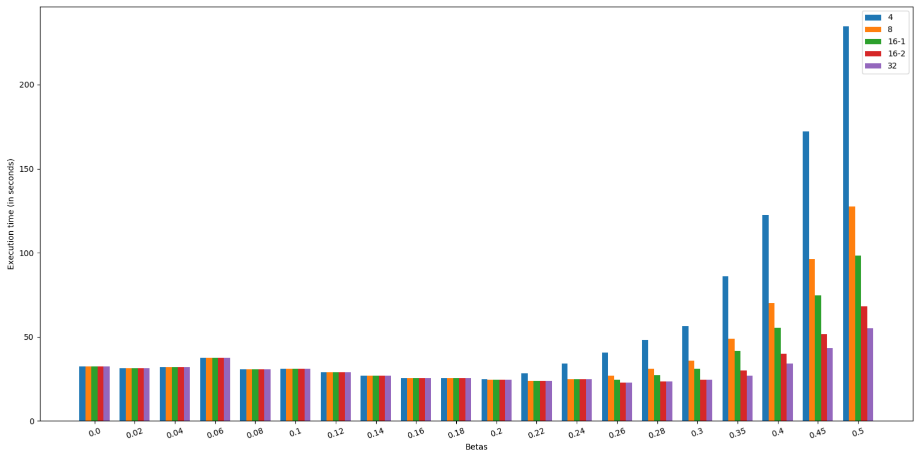


Figure 4.11: Execution times for the GEMM experiment with varying CPU threads.

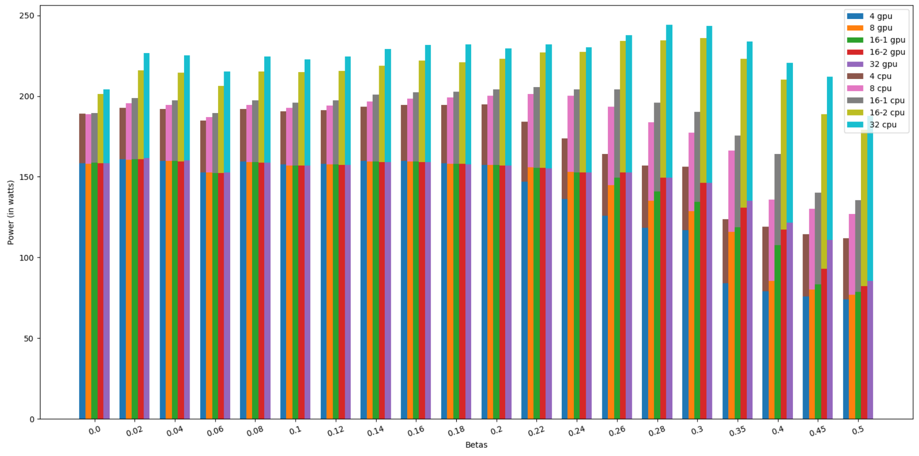


Figure 4.12: Total power draw for the GEMM experiment with varying CPU threads.

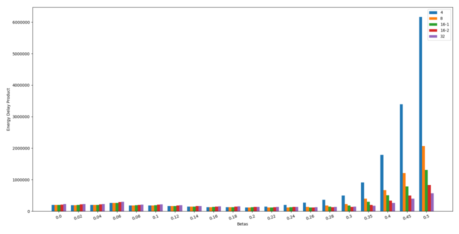
the workload between the two sockets (the DM CPU has two sockets), where each socket uses 8 threads to execute the workload.

Figure 4.11 shows the execution time using these different configurations. We observe that, the more threads the application has at its disposal, the lower the execution time. This is obvious as the threads we would not use, would otherwise be idle. Interestingly, the two configurations where we use 16 threads are different in execution time. The difference is that 16-1 uses 8 cores with two hardware threads per core, while 16-2 uses 16 ”real” cores. This shows that the performance of hyper-threading is not the same as that of actual cores.

This experiment also demonstrates that the optimal β differs between configurations: the optimal β is around 20% for 4 threads, but it increases to around 26% for the 16-2 and 32 threads configurations. However, our analysis methodology holds for all these different configurations. We note also that the curve stays the same, and only the optimal β changes. We see that when using the maximum amount of resources available, we achieve the lowest execution times, most notable for larger β values.

In Figure 4.12 we illustrate the average power draw. We see the power draw stays high for the 16-2 and 32 threads configurations. This is because of the lower execution times of the CPU, which means the GPU is idling less. Contrary to when we use only 4 threads, then the GPU is idling a lot more and has a lot of time to cool down.

Figure 4.13 shows the energy efficiency in terms of Energy Delay Product and Flops per watt for varying CPU threads. We observe that the usage of more threads is better for the energy efficiency, because lowering the overall execution time indirectly improves the energy efficiency.


(a) Energy Delay Product

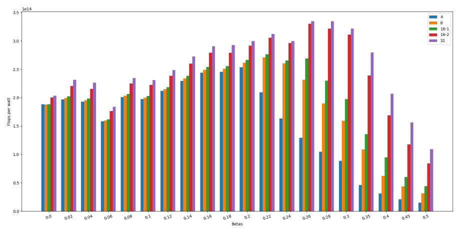

(b) FLOPS per watt

Figure 4.13: Energy efficiency in terms of Energy Delay Product and FLOPS per watt for the GEMM experiment with varying CPU threads.

We notice that the GPU is still considered more energy efficiency with both metrics.

Finally, we also note that the difference for the CPU using one or both sockets is significant in terms of power draw: the 16-2 configuration is much more expensive, power-wise, than the 16-1 version.

4.5 Summary

For all studied applications, we were able to empirically identify the best performing workload partitioning, and found this optimal partitioning to coincide with the most energy efficient partitioning. The total power draw follows the same trend in all tested benchmarks, indicating idle energy is a big problem when GPU and CPU have different execution times.

The different benchmarks do show the different impact of heterogeneous computing. The reduction and stencil1D show little to no speedup compared to their homogeneous GPU versions. This questions the need for heterogeneous versions of these applications. The GEMM and Stencil2D show speedup in their heterogeneous versions, due to the increased arithmetic intensity of these benchmarks.

Finally, the two different metrics we use for energy efficiency highlight different aspects. We see that a slightly imbalanced workload towards the GPU does not affect EDP significantly; this is an indirect proof of the extra performance and efficiency of the GPU. When looking at FLOPS per watt, we observe that workload imbalance might lead to bigger energy efficiency losses, but this is dependent on the application itself (as expected, as the number of FLOPs is an application characteristic).

CHAPTER 5

Related Work

In this chapter we briefly present related work that combines energy efficiency with heterogeneous systems, and discuss other work where energy played a big part in heterogeneous systems.

There are research efforts that focus on lowering the energy consumption of GPU+CPU architectures, but they address either the GPU or the CPU in isolation, and thus cannot achieve maximized energy savings. An approach for holistic energy management of CPU+GPU computations is presented in [1] by the name of GreenGPU. The solution features a two tier design. In the first tier, GreenGPU dynamically splits and distributes workloads to GPU and CPU based on the workload characteristics, such that both sides can finish approximately at the same time. As a result, the energy wasted on idling and waiting for the slower side to finish is minimized. In the second tier, GreenGPU dynamically throttles the frequencies of GPU cores and memory in a coordinated manner, based on their utilization's, for maximized energy savings with only marginal performance degradation. Likewise, the frequency and voltage of the CPU are scaled similarly. The authors claim to be able to save 20% on average.

Qiang Liu and Wayne Luk [24] explore heterogeneous computing hardware to maximize system energy efficiency while considering system performance and power consumption. The approach presented could help system designers to evaluate and choose the right combinations of high energy-efficient devices and interfaces, when designing energy efficient scientific computing systems. The sensitivity of the system energy efficiency to individual system components is also studied. Their results indicate that the improvement of system energy efficiency is more sensitive to some of the system components than to others. For example, in the studied system, concurrently improving the energy efficiency of the interface and the GPU by 10 times could lead to over 10 times improvement of the system energy efficiency.

There are various parallel programming frameworks such as: OpenMP, OpenCL, OpenACC and CUDA. Selecting the one that is suitable for a target context is not straightforward. Suejb Memeti et al. [25] have empirically studied the characteristics of OpenMP, OpenACC, OpenCL, and CUDA with respect to programming productivity, performance, and energy. For comparison they used the SPEC Accel and Rodinia benchmark suites on two heterogeneous computing systems. With regard to performance and energy consumption they note that less execution time usually leads to less energy consumption.

Efficient application scheduling is critical for achieving high performance in heterogeneous computing environments. Tarplee et al. [26], show a solution for optimizing two conflicting objectives. The first is to minimize the makespan, that is, the maximum finishing time of all tasks. The second is to minimize the total energy consumption of all machines in a heterogeneous computing system. They used a linear programming method to generate a set of high-quality solutions that represent the trade-off space between makespan and energy consumption. Their scheduling algorithm is highly scalable to schedule tasks in heterogeneous computing systems. The heterogeneity in execution time of the tasks provides the scheduler degrees of freedom to greatly improve the performance as compared to a naive scheduling algorithm. Similarly the heterogeneity in the power consumption allows the scheduler to decrease the energy consumption.

Li et al. [27], study the problem of scheduling a bag-of-tasks application, made of a collection of independent stochastic tasks with normal distributions of task execution times on a heterogeneous platform with deadline and energy consumption budget constraints. Their algorithm does not guarantee optimal performance - instead, the main objective is to improve the weighted probability that both a deadline and an energy consumption budget constraints are met.

To address the surging demand for computing capacity, more and more heterogeneous systems have been designed. The huge energy consumption of these heterogeneous systems raises new environmental concerns. Besides performance, energy efficiency is now another key factor to be considered by system designers and also consumers. Wang et al. [28] present EPPMiner a benchmark suite for evaluating the performance, power, and energy of different heterogeneous systems. Bombieri et al. [29] present MIPP, a suite of microbenchmarks that aim at characterizing a GPU device in terms of performance, power, and energy consumption. MIPP aims at understanding how application bottlenecks involving selected functional components or underutilization of them can affect code performance, power consumption, and energy efficiency on a given device. Che et al. [30] developed Rodinia a benchmark suite for heterogeneous computing and is a standard benchmark to compare platforms. Its applications are inspired by Berkeley's dwarf taxonomy and they have developed kernels to run on multi-core CPUs (OpenMP), GPUs (CUDA) and OpenCL.

In summary, our research was inspired by previous work that also weighted the performance against energy consumption in the case of heterogeneous computing, like [1] and [26]. However, our work has analysed performance and energy efficiency from a different perspective: our focus was on the tension between performance and energy consumption for multiple applications, while most other work focuses on optimizing the performance of a specific application, with energy consumption as a constraint.

CHAPTER 6

Conclusion and Future Work

In search for high-performance computing, many applications turn to multi-core CPUs and/or GPUs for performance. However, CPUs and GPUs have been developed with different kinds of applications in mind - i.e., they have been optimized to perform different tasks. Therefore, heterogeneous CPU+GPU architectures could provide a solution for many complex applications, that combine CPU- with GPU-specific phases. Optimizing application performance for such heterogeneous platforms is a well studied problem. However, a lot less studies focus on the energy efficiency of these systems.

In this thesis we have studied how the use of heterogeneous platforms affects the energy consumption of the composing PUs, and whether there is a difference in optimizing for performance and energy efficiency on such platforms.

6.1 Main Findings

Our research was based on one research question:

is there tension between performance and energy efficiency in heterogeneous computing?

We have devised an empirical method to confirm whether this tension exists. Our results show the effect of heterogeneous computing for three different applications, concluding that heterogeneous computing does not bring only positive changes for every application. Heterogeneous computing requires more work for the programmer and results in more difficult analysis of the problem. For the applications with a high arithmetic intensity, we have shown that heterogeneous computing does offer a solution, resulting in lower execution times.

From our results we can further conclude that the energy consumption is strongly correlated with the execution time of the applications. We observed that the curve for the total energy consumption and execution times of all our tested benchmarks follow the same trends. Thus, for an optimal workload partitioning we found both the lowest execution time and lowest energy consumption for a benchmark. This means that if one is trying to optimize in terms of performance, they would indirectly optimize the energy consumption of the application as well.

To summarize, we conclude there is no tension between performance and energy efficiency for the studied CPU+GPU heterogeneous platforms, since we observe there is no loss in performance when energy consumption comes first, and vice versa.

6.2 Future Work

There are directions we envision to continue this research.

First, we can expand the number and type of tested applications. Besides the benchmarks we have used in this thesis, there are many more to be chosen from the SHOC [13] benchmark suite. There also exist other benchmark suites, like for example Parboil and Rodinia, which are GPU benchmarks comprised of scientific applications. These benchmarks also provide implementations for the OpenCL programming model, an area we have left completely unexplored throughout the entirety of this thesis.

Second, we have only used one method to measure energy consumption of heterogeneous applications. Existing studies, like [9], list more popular techniques to measure energy efficiency. These techniques could be employed in future research to bolster and verify our results.

Third, accuracy can be further boosted. Through the findings in our study we can not recommend the on-chip sensors, RAPL for CPUs and NVML for GPUs. The main issue with the measurements is the lack of information about how the power reading is performed for such a component. In order to produce more reliable results we urge hardware vendors to further develop their on-chip sensor technology. In order to provide scientists with necessary information of how a power measurement is determined for a component, the sampling rate of the measurements, its reported accuracy and finally how to obtain this measurement with sufficient accuracy and low overhead.

Finally, further specialization could be done in the form of testing the use of multiple CPUs and multiple GPUs. Using a single GPU with multiple CPUs requires specific software and very specific applications, as those applications should not contain massive parallelism, to enable the time-sharing of the single GPU. The alternative option, using multiple GPUs with a single CPU, is also left unstudied. It is likely this configuration can be studied as a generalization of the already studied CPU+GPU configuration in this thesis.

Bibliography

[1] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, “Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures,” in *2012 41st International Conference on Parallel Processing*, IEEE, 2012, pp. 48–57.

[2] A. S. Andrae and T. Edler, “On global electricity usage of communication technology: Trends to 2030,” *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.

[3] W.-c. Feng, X. Feng, and R. Ge, “Green supercomputing comes of age,” *IT professional*, vol. 10, no. 1, pp. 17–23, 2008.

[4] (1993). The top 500 list, [Online]. Available: `https://www.top500.org/`.

[5] (2007). The green 500 list, [Online]. Available: `https://www.top500.org/green500/`.

[6] (2019). Exploring the gpu architecture, [Online]. Available: `https://nielshagoort.com/2019/03/12/exploring-the-gpu-architecture/`.

[7] OpenMP Architecture Review Board. (2015). OpenMP application program interface version 4.5, [Online]. Available: `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

[8] (2019). The cuda c++ programming guide, [Online]. Available: `https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf`.

[9] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, “A comparative study of methods for measurement of energy of computing,” *Energies*, vol. 12, no. 11, p. 2204, 2019.

[10] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, “Rapl in action: Experiences in using rapl for power measurements,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, pp. 1–26, 2018.

[11] (2017). Energy consumption measurement of c/c++ programs using clang tooling, [Online]. Available: `http://ceur-ws.org/Vol-1938/paper-san.pdf`.

[12] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, “Measuring energy consumption for short code paths using rapl,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[13] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The scalable heterogeneous computing (shoc) benchmark suite,” in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.

[14] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[15] M. Harris *et al.*, “Optimizing parallel reduction in cuda,” *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.

[16] A. Mahardito, A. Suhendra, and D. T. Hasta, “Optimizing parallel reduction in cuda to reach gpu peak performance,” *Skripsi Program Studi Sistem Komputer*, 2010.

[17] (2018). Matrix-matrix multiplication on the gpu with nvidia cuda, [Online]. Available: `https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/`.

[18] (2020). Chrono: Date and time utilities, [Online]. Available: `https://en.cppreference.com/w/cpp/chrono`.

[19] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *2010 39th International Conference on Parallel Processing Workshops*, IEEE, 2010, pp. 207–216.

[20] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, “Measuring energy and power with papi,” in *2012 41st International Conference on Parallel Processing Workshops*, IEEE, 2012, pp. 262–268.

[21] (2019). Nvml reference manual, [Online]. Available: `https://docs.nvidia.com/pdf/NVML_API_Reference_Guide.pdf`.

[22] J. S. Vetter, *Contemporary high performance computing: from Petascale toward exascale.* CRC Press, 2013.

[23] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, “A medium-scale distributed system for computer science research: Infrastructure for the long term,” *Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[24] Q. Liu and W. Luk, “Heterogeneous systems for energy efficient scientific computing,” in *International Symposium on Applied Reconfigurable Computing*, Springer, 2012, pp. 64–75.

[25] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, “Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption,” in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 2017, pp. 1–6.

[26] K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. Chong, “Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques,” *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 6, pp. 1633–1646, 2015.

[27] K. Li, X. Tang, and K. Li, “Energy-efficient stochastic task scheduling on heterogeneous computing systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 2867–2876, 2013.

[28] Q. Wang, P. Xu, Y. Zhang, and X. Chu, “Eppminer: An extended benchmark suite for energy, power and performance characterization of heterogeneous architecture,” in *Proceedings of the Eighth International Conference on Future Energy Systems*, 2017, pp. 23–33.

[29] N. Bombieri, F. Busato, F. Fummi, and M. Scala, “Mipp: A microbenchmark suite for performance, power, and energy consumption characterization of gpu architectures,” in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, IEEE, 2016, pp. 1–6.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*, Ieee, 2009, pp. 44–54.

APPENDIX A

Sample Output

NVIDIA-smi reports the queried properties back every specified timestep, in this case every second. The output then consists of a line (here displayed as a table) with all properties split by a comma. The first line can removed with the noheader specification.

timestamp	fan.speed	utilization.gpu	utilization.memory	temperature.gpu	power.draw
09:18:39.668	25 %	0 %	0 %	48	65.36 W
09:18:40.670	25 %	1 %	0 %	48	65.29 W
09:18:41.677	25 %	0 %	0 %	48	65.49 W
09:18:42.679	25 %	0 %	0 %	48	65.39 W
09:18:43.682	25 %	0 %	0 %	48	65.26 W
09:18:44.684	25 %	0 %	0 %	48	65.40 W
09:18:45.687	25 %	0 %	0 %	48	65.46 W
09:18:46.689	25 %	0 %	0 %	48	65.29 W
09:18:47.692	25 %	0 %	0 %	48	65.36 W
09:18:48.698	25 %	0 %	0 %	48	65.49 W
09:18:49.706	25 %	0 %	0 %	48	65.55 W
09:18:50.713	25 %	0 %	0 %	49	65.58 W

Table A.1: Example output from nvidia-smi for a 10 second sleep experiment.

The LIKWID output consists of a table with the raw event counts and another table with derived metrics. We can optionally output easily parseable CSV instead of the fancy tables. The columns are the processor ids measured. If more than one core is measured, there is another table with statistical data like sum, minimum, maximum and average of all measured cores.

Listing A.1: Example output from likwid-perfctr for a 10 second sleep experiment, shortened to four out of sixteen cores.

```
--------------------------------------------------------------------------------
CPU name:       AMD Ryzen 7 1800X Eight-Core Processor
CPU type:       AMD K17 (Zen) architecture
CPU clock:      3.59 GHz
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Group 1: ENERGY
+-----------------------+---------+----------+----------+-----------+----------+
|         Event         | Counter |  Core 0  |  Core 1  |   Core 2  |  Core 3  |
+-----------------------+---------+----------+----------+-----------+----------+
|   ACTUAL_CPU_CLOCK    |  FIXC1  | 43652465 | 10869343 |  62933754 | 17985558 |
|     MAX_CPU_CLOCK     |  FIXC2  | 82339020 | 20552688 | 119226204 | 34307928 |
| RETIRED_INSTRUCTIONS  |   PMC0  | 15954480 |  1391734 |  14650221 |   272459 |
|  CPU_CLOCKS_UNHALTED  |   PMC1  | 26994482 |  3345522 |  36892760 |  2759589 |
|    RAPL_CORE_ENERGY   |   PWR0  |   0.1846 |        0 |    0.1822 |        0 |
|    RAPL_PKG_ENERGY    |   PWR1  | 157.7101 |        0 |         0 |        0 |
+-----------------------+---------+----------+----------+-----------+----------+

+----------------------------+---------+------------+---------+------------+-------------+
|            Event           | Counter |     Sum    |   Min   |     Max    |     Avg     |
+----------------------------+---------+------------+---------+------------+-------------+
|   ACTUAL_CPU_CLOCK STAT    |  FIXC1  | 2298410883 | 4066225 |  597592356 | 1.43650e+08 |
|     MAX_CPU_CLOCK STAT     |  FIXC2  | 4143424536 | 7747596 | 1070531820 | 2.58964e+08 |
| RETIRED_INSTRUCTIONS STAT  |   PMC0  |  668419954 |   43028 |  180134582 | 4.17762e+07 |
|  CPU_CLOCKS_UNHALTED STAT  |   PMC1  | 1047150183 |  323393 |  255150014 | 6.54468e+07 |
|    RAPL_CORE_ENERGY STAT   |   PWR0  |     3.5431 |       0 |     0.7646 |      0.2214 |
|    RAPL_PKG_ENERGY STAT    |   PWR1  |   157.7101 |       0 |   157.7101 |      9.8569 |
+----------------------------+---------+------------+---------+------------+-------------+

+-----------------------+-----------+-----------+-----------+-----------+
|        Metric         |   Core 0  |   Core 1  |   Core 2  |   Core 3  |
+-----------------------+-----------+-----------+-----------+-----------+
|  Runtime (RDTSC) [s]  |    9.9998 |    9.9998 |    9.9998 |    9.9998 |
| Runtime unhalted [s]  |    0.0121 |    0.0030 |    0.0175 |    0.0050 |
|      Clock [MHz]      | 1904.9598 | 1900.2791 | 1896.6826 | 1883.7018 |
|          CPI          |    1.6920 |    2.4039 |    2.5182 |   10.1285 |
|    Energy Core [J]    |    0.1846 |         0 |    0.1822 |         0 |
|    Power Core [W]     |    0.0185 |         0 |    0.0182 |         0 |
|    Energy PKG [J]     |  157.7101 |         0 |         0 |         0 |
|     Power PKG [W]     |   15.7714 |         0 |         0 |         0 |
+-----------------------+-----------+-----------+-----------+-----------+

+----------------------------+------------+-----------+-----------+-----------+
|           Metric           |     Sum    |    Min    |    Max    |    Avg    |
+----------------------------+------------+-----------+-----------+-----------+
|  Runtime (RDTSC) [s] STAT  |   159.9968 |    9.9998 |    9.9998 |    9.9998 |
| Runtime unhalted [s] STAT  |     0.6396 |    0.0011 |    0.1663 |    0.0400 |
|      Clock [MHz] STAT      | 31132.1189 | 1881.9444 | 2071.9843 | 1945.7574 |
|          CPI STAT          |    46.9679 |    1.2785 |   10.1285 |    2.9355 |
|    Energy Core [J] STAT    |     3.5431 |         0 |    0.7646 |    0.2214 |
|    Power Core [W] STAT     |     0.3544 |         0 |    0.0765 |    0.0221 |
|    Energy PKG [J] STAT     |   157.7101 |         0 |  157.7101 |    9.8569 |
|     Power PKG [W] STAT     |    15.7714 |         0 |   15.7714 |    0.9857 |
+----------------------------+------------+-----------+-----------+-----------+
```