Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 120

Dynamic workload balancing for heterogeneous systems

Alexander Strack

Course of Study:	B.Sc. Simulation Technology
Examiner:	Prof. Dr. Miriam Mehl
Supervisor:	M.Sc. Malte Brunn
Commenced:	June 01, 2020
Completed:	October 21, 2020

Abstract

During the last two decades, GPUs developed into powerful and massively parallel processors. That rose the attention of scientist who started using GPUs for large scale scientific computing, e.g. simulations. However, the architecture of GPUs is different from CPUs. Furthermore, graphic processors have their now fast access memory. Computing in a heterogeneous system consisting of a CPU and multiple GPUs has various challenges. In this work, we focus on how to distribute the load among the different components. We consider an iterative load that can be redistributed after each iteration. The goal of our scheduling methods is to minimise the computation time of the next iteration by estimating the performance of each component. After a short introduction to load balancing, we specify the iterative workload scenario and differentiate it from the typical task-based scenario often found in the literature. Then, we show the basics of GPU programming with the help of NVIDIAs CUDA API. Furthermore, we introduce the different kernels we use for our test and derive multiple schedulers. Our dynamic schedulers use the time each component took to compute its assigned workload in the last iteration as a basis of the performance estimation. After investigating the influence of previous run-time data on the scheduling decisions, we turn our attention towards the properties of the workloads and therefore compare different types of memory management.

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

Computing hardware shifted from single processors with one core to complex systems with many multi-core processors. These processors can be located at one place or distributed across the globe and connected via the internet. The challenge of using these heterogeneous systems as effective as possible developed into the broad topic of load balancing. In this work, we focus on small heterogeneous systems that consist of a CPU and multiple GPUs and try to dynamically, iteratively balance a workload.
The first two chapters are part of the propadeuticum required for the Simulation Technology degree while all following chapters are part of the corresponding bachelor thesis. In the first chapter, we recapitulate the development of PC hardware and talk about the motivation for this thesis. Next, we give a short introduction to load balancing and show the necessary basics of GPU programming with the CUDA API. Then, as the first part of the bachelor thesis, we introduce several kernels as well as the scheduling algorithms we used for benchmarking. In Chapter 4 we formulate our different tests and estimate the theoretical performance based on the properties of the kernel chosen as workload. After validating our estimation with the results of our tests, we draw a conclusion, answer our motivating questions and give a short future look.

1.1 Historical background

The journey of the microprocessor starts in 1971 when Intel released the 4004. This first microprocessor was originally designed for a calculator, clocked at 740kHz, and housed 2300 transistors. In the following years, people began to see the potential microprocessors had to revolutionize computing. During the third quarter of the 20th century, the performance of microprocessors rapidly improved. Back in 1965 this rise was astonishingly well predicted by the Intel co-founder Gordon E. Moore. He predicted a doubling of the transistor count in processors every twelve months until 1975 [Moo65]. Ten years later he changed his prognosis to a doubling every 24 months [Moo75]. This prediction is still valid until today as illustrated in Figure 1.1.
But it was not shortly before the turn of the millennium when AMD launched the first microprocessor clocked at 1GHz, beating its rival Intel only by a few days. Around the same time, AMD won the clock speed battle to 1GHz, NVIDIA released the GeForce 256.

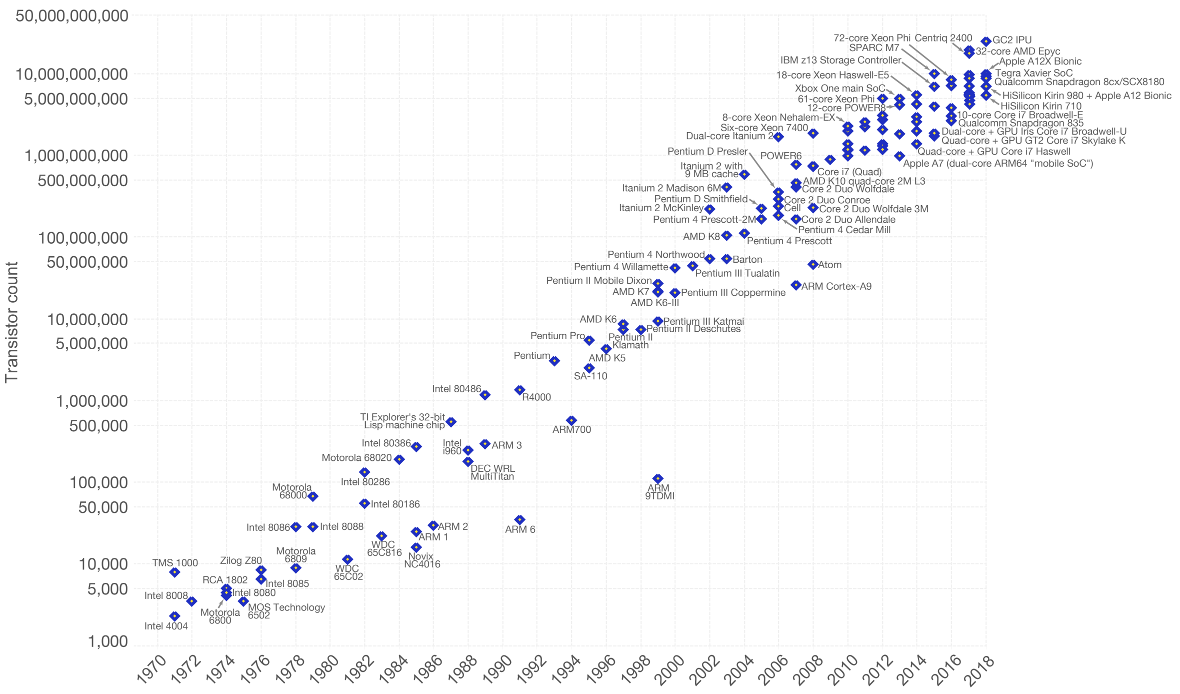


Figure 1.1: Transistor count development in the last 50 years. Each point marks a processor e.g. the Intel 4004 released in 1971 with 2300 transistors or the Intel Pentium released in 1993 with over 3 million transistors. Gordon E. Moore predicted in 1975 a doubling of the transistor count in integrated circuits approximately every two years. It became famous as "Moore's law" and is a fundamental and astonishingly accurate prediction of technological progress [Ros19].

This was the first GPU as we know it today and was capable of image transformation which maps 3D visuals on a regular 2D monitor. That task was previously left for the more powerful CPU. In the following years, the computation power of NVIDIA and ATI GPUs rapidly increased. The rise of the accelerators began.
During the last 20 years, the theoretical peak FLOPS (floating-point operations per second) performance of CPUs and GPUs evolved differently (see Figure 1.2). At the same time, the memory bandwidth of GPUs skyrocketed (see Figure 1.3). While CPU development mainly focused on high clock speed on a few cores, GPUs are clocked at a lower speed but have many more cores. The differences in chip design are caused by the different properties of the tasks they are designed for. A CPU has to execute a few threads as fast as possible so high clock speed but not many cores are required. In contrast, a GPU has to do many tasks in parallel. This led the development to many cores with lower clock speed. The differences between the two architectures are illustrated in Figure 1.4. Furthermore, GPUs use SIMD-instructions which provide additional parallelism. So GPUs are comparable to CPUs with AVX extensions. Though, GPUs use typically a greater vector length.

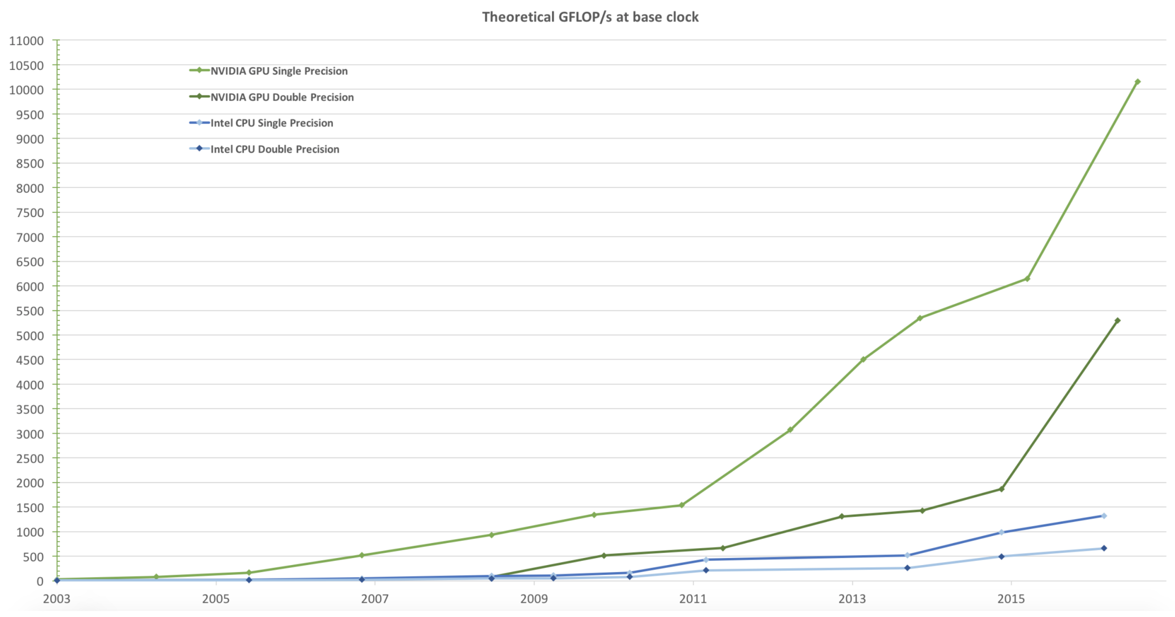


Figure 1.2: Theoretical FLOPS performance development of Intel CPUs and NVIDIA GPUs. The performance of Intel CPUs (blue) massively increased during the last two decades. However, the rise is dwarfed by the performance increase NVIDIA GPUs (green) experienced in the same period. Due to a different architecture with more cores but lower clock speed GPUs outperform its few cores, high clock speed counterparts by a factor of up to 7 in raw computational performance. Even though GPUs are not designed for double-precision computations they still outperform CPUs by roughly the same factor [Nvib].

Because of lower clock speeds, the power consumption of modern GPUs is only slightly higher than the power consumption of CPUs. The much higher instruction throughput and memory bandwidth allow GPUs to work with much more data. This is beneficial for typical graphic applications e.g. rendering. But the high level of parallelism is also usable for scientific computations e.g. large scale simulations, where we need to perform the same steps for each grid node. So with the rise of computing power also rose the interest of scientists in GPU computing.

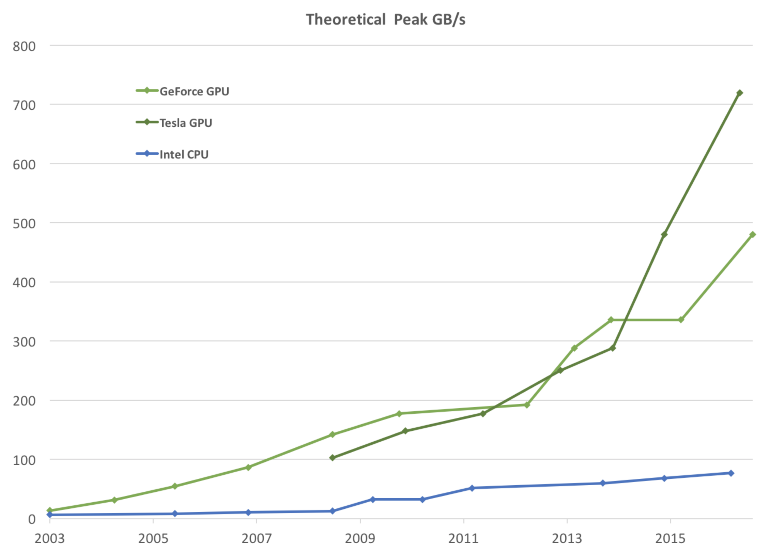


Figure 1.3: Theoretical memory bandwidth development of Intel CPUs and NVIDIA GPUs. The bandwidth of hardware components increased rapidly over the last two decades. However, CPUs (blue) are still not able to cross the 100GB/s mark. GPUs (green) in contrast have beaten the mark over a decade ago and have currently up to ten times greater bandwidth [Nvib].

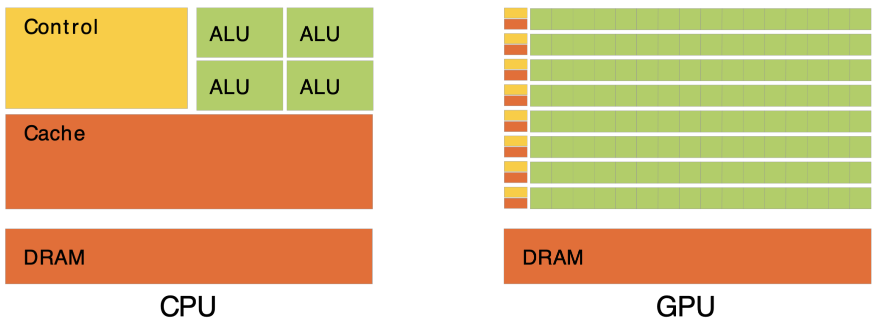


Figure 1.4: The chip architecture of CPUs and GPUs is different. A CPU focuses on only a few arithmetic logic units (ALUs) and uses a big control block to handle many different instructions. The cache block is divided into multiple levels and crucial for fast data access on CPUs. Cache is much faster than any form of DRAM but in comparison very small. In contrast to this versatile approach, GPUs use almost the entire space to accommodate many ALUs [Nvib].

1.2 Motivation

In the last section, we showed the exponential performance growth of GPUs in the last 20 years. Not only video editors or gamers profited from it. Among scientists, it also became popular to use GPUs to accelerate computations. However, for the GPU to work in hand with the CPU, we have to consider inter alia the following questions:

- How do we distribute the workload among the CPU and GPU to exploit the advantages of both architectures?
- Is the outsourcing of work to a GPU useful or does the communication negate the performance gain?
- What influence have the workload properties and the memory management on the workload distribution?

All of them are part of the topic of load balancing. Therefore, we will shortly introduce load balancing and give a general overview of different load balancing strategies. Starting from a general viewpoint we lead to the specific load balancing scenario investigated in this thesis.

The typical load balancing approach is based on scheduling tasks e.g. in [Hag97] or [Bin13]. In addition, task scheduling is the main load balancing topic on distributed systems see [LR91], [MJ14] and [Sah13] or in cloud computing [KM14] and [MR16]. However, we focus on balancing an iterative load that is scheduled after each iteration to minimize the computation time. In this research area, [ABA12] and [ABA13] contributed work on multi-GPU systems. Basis of their work was the *ULL_Calibrate_lib* [GABC08] which is a scheduling library providing minimal computation overhead and is easy to use. Iterative load balancing on distributed systems was treated by [XL95] and [BCV09].
A essential part of each load balancing algorithm is to make a good scheduling decision. [DL15] use an approach where the computation time of a task is predicted by profiling codes. In contrast, [GBHS12] scheduler uses previous run-time data to estimate the run-time of the currently scheduled tasks. [Boy13] proposed a scheduler that first assigns small partitions of the workload to the nodes and increases the partition size each time a node has finished the previous partition. So there exist various approaches on how to estimate the system performance. Even a fuzzy neural network was used by [ZXZ$^+$17]. The scheduling approach we use is based on the time each component took to compute its assigned workload in the last iteration. For a more detailed explanation see Section 3.2.

CHAPTER 2

Basics

This chapter contains a short introduction to load balancing and a definition of the iterative balancing scenario. Furthermore, we introduce the fundamental basics of GPU programming with the CUDA API provided by NVIDIA.

2.1 Load balancing

Many more or less different types of load balancing exist. To get a general overview, we classify the different types with the help of the frequently used taxonomy of Casavant and Kuhl [CK88]. Furthermore, we explain the different hierarchy levels of the taxonomy. Then, we give an example of a typical task-based dynamic load balancing scenario and derive an iterative scenario that better represents scientific simulations.

2.1.1 Taxonomie

There exist various schemes to classify load balancing. The most popular is the taxonomy proposed by Casavant and Kuhl [CK88]. A simplified version of the taxonomy can be seen in Figure 2.1. The following explanation style was inspired by [SNO+11]. We refer to a part of the system as node or component. On the first level, the model divides scheduling approaches into local and global strategies.

Local versus global A local scheduler only distributes the workload on one node. It is not dependent on any other node in the system. Whereas a global scheduler uses information about all nodes in the system to allocate tasks to different nodes. Its goal is to optimize a global performance objective e.g. node utilization. Compared to global scheduling, local scheduling requires no communication between nodes but is a dead end when we have a system with multiple nodes and want to optimize a global performance objective. As we focus on global schedulers, the next hierarchy level is static versus dynamic.

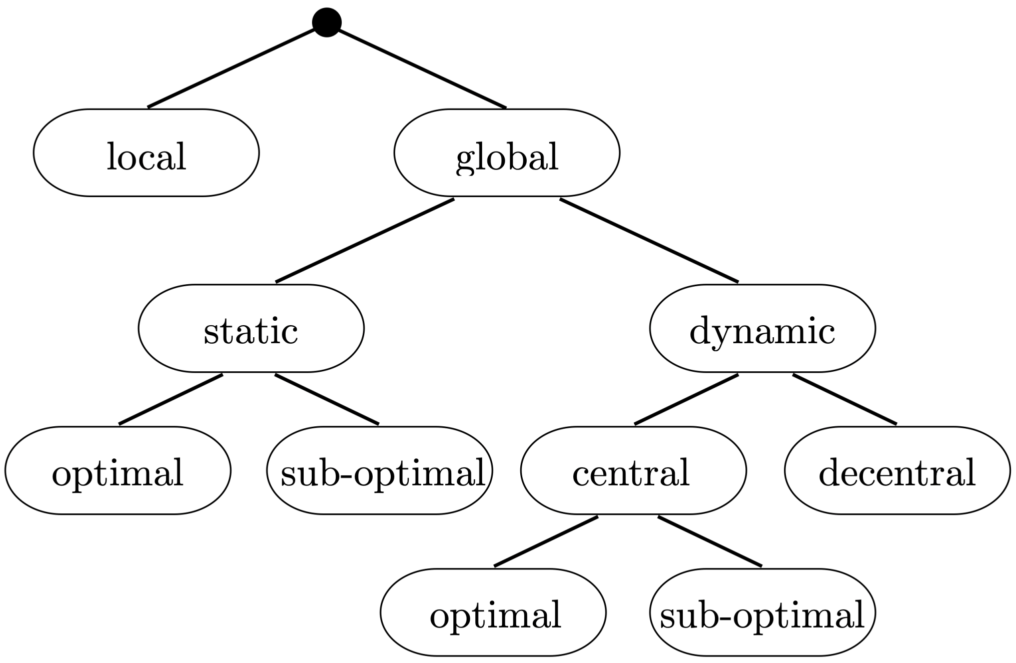


Figure 2.1: The load balancing taxonomy by Casavant and Kuhl [CK88]. This version was modified from the original and contains only new leaves where a lower level of the hierarchy is relevant for this thesis. For a more detailed explanation of the different hierarchy levels see Section 2.1.1.

Static versus dynamic In static scheduling, we assume to have all the necessary information at hand when we make the scheduling decision and then never change it during the entire run-time. We need information about the computation time of the task at each node. Further, we need to know the communication cost and structure of the system. For each task, a static scheduler makes only one decision before the application is running. In contrast, a dynamic scheduler makes various scheduling decisions during the run-time. Therefore, it has some advantages compared to static scheduling. For example, the computation time of a finished task can be used to improve the next decision because the performance of the node can be better approximated. Furthermore, a dynamic scheduler can handle tasks that are created while the application is running. Another benefit of a dynamic scheduler is its resistance against sudden performance changes of certain nodes e.g when they have to compute tasks for another application. However, dynamic approaches generate additional cost and make the application more complex. So choosing a dynamic scheduler is not always the optimal solution.

In a homogeneous system, static scheduling is attractive because we can assume the same execution time on each node and assign the tasks equally. Considering a heterogeneous system, this type of static scheduling is not efficient because to get good results we must somehow approximate the performance of each node before the application run-time.

Central versus decentral Dynamic approaches can be further divided into centralized and decentralized scheduling. In a scenario, with a central scheduler one node - typically referred to as host - collects all the system information and makes the scheduling decisions. This has the advantage that all the information is accumulated at the host and available for the scheduling decisions. In general, that leads to better decision making compared to a decentralized scheduler. Furthermore, it is easier to implement because all other nodes do not have to deal with scheduling. But it lacks scalability for very large or distributed systems. In such systems, it may be more effective to only have some information e.g. about the neighbour nodes, and request or send tasks to them.

Optimal versus sub-optimal As mentioned in the last paragraphs global schedulers need information about all nodes and the unscheduled tasks. Especially static schedulers need all the information about the system before the run-time. Because scheduling is an NP-complete problem [Ull75] and it is difficult to make general performance assumptions, static schedulers found in the literature are typically sub-optimal and use heuristic or approximative models to make scheduling decisions. The optimization problem can be solved e.g. by gradient descent or heuristic methods like Simulated Annealing [Pol99] or Genetic Models [SP94].

2.1.2 Iterative load balancing

The typical task-based dynamic load balancing scenario is the following:

- Dynamic centralized sup-optimal scheduler $\rightarrow$ one host accumulates the entire system information and makes the scheduling decision
- Tasks have different types and sizes and are typically scientific applications
- Tasks are generated by a Poisson process and get scheduled on-the-fly
- The host is only responsible for scheduling and does not execute tasks - even if idle
- Goal: maximize the utilization of all nodes

Most scientific simulations do some sort of discretization in space and time. Thus, the spatial is divided into a grid of size N points and there is a time step size of Δt. Each time step t_i is Δt bigger than its predecessor. The resulting task has always size N and is iterated over the time steps t_i. This is why we investigate an iterative scheduling scenario.

The scheduler is like in the task-based scenario dynamic, centralized, and sup-optimal. The goal is to minimize the computation time of one iteration. In contrast to the task-based scenario where the host node is only responsible for scheduling, the host assigns itself a part of the static task to which we will refer as workload from now on. That is possible because scheduling decisions only have to be made at the start of each iteration. Furthermore, will we refer to our non-host nodes in the system as devices because in our test systems they are all GPUs while the host is the only CPU. So our heterogeneous system consists of one CPU (host) and n GPUs (devices). The scheduling approach is to redistribute the workload on the host and devices in such a way that the new distribution fits the current performance estimation of the system. So our approach impacts a lower level of parallelism as we do not schedule tasks but try to partition one iterated task optimally. In fact, we only parallelize a kernel and redistribute the partitions on which the nodes operate. In the next chapter, we introduce the kernels used as workload and the scheduling algorithms we developed for our benchmarks.

2.2 GPU programming with CUDA

This section contains a short introduction to the CUDA API which enables scientific computing on NVIDIA GPUs. Released in 2007, CUDA (Compute Unified Device Architecture) is an NVIDIA GPU exclusive API. CUDA can be wrapped for many programming languages e.g. Python or MATLAB. But the conventional way to use it is "C for CUDA" which simply extends the language C with specific commands and functions. In the following paragraphs, we show how to write code for GPUs, launch CUDA-kernels, allocate memory, copy data and measure the execution time of CUDA-kernels.

Core of the C extension are CUDA-kernels i.e. functions that run exclusively on the device. As mentioned above, we refer to the CPU as host and to GPUs as devices. In code, a CUDA-kernel can be identified by the declaration specifier *__global__*. To call a CUDA-kernel we need to add a $<<< X, Y >>>$ expression (see Algorithm 2.1). These two parameters specify the number of blocks and on how many threads per block the code is executed in parallel. Typically the number of threads per block is limited to 1024 on current hardware. The maximal number of blocks is way larger. Furthermore, the threads and blocks can be organized in up to three dimensions. Figure 2.2 gives an example for a computation grid containing a two-dimensional set of six blocks, with twelve threads each. When called, a CUDA-kernel is executed asynchronously by the device. To synchronize the kernel e.g. when the results of the kernel are required for the host to proceed, use the method *cudaDeviceSynchronize()*. With this routine, the host is forced to wait until all CUDA-kernels are finished.
To make a performance-based scheduling decision we need to measure the time the device needs to execute a kernel. This measurement has to be synchronous and can be implemented with a CUDA provided tool called events. To measure the execution time of a kernel we have to create two events - start and stop.

Algorithm 2.1 CUDA-kernel call and time measurement

The algorithm shows the declaration of a CUDA-kernel. Furthermore, it shows how to call the kernel with the specifiers X, Y that indicate the number of blocks and threads per block. The kernel is timed with the help of two CUDA-events.

```
__global__ void kernel( void ) {}
int main( void )
{
  cudaEvent_t start , stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);
  kernel<<<X, Y>>>();
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  float elapsedTime;
  cudaEventElapsedTime(&elapsedTime, start, stop)
  return 0;
}
```

Then, we record the start event before and the stop event after the kernel call and compute the time difference via the provided function *cudaElapsedTime()*. This kind of time measurement requires the host thread to wait until the CUDA-kernel finished. During that period, the host cannot compute its part of the workload. Unfortunately, the execution times are crucial for our performance estimation. At first glance, a parallel kernel execution of host and device seems not feasible. But there is a quite simple solution to this problem. The host thread is only capable of handling one device [SK11]. Therefore, a n device system requires $n-1$ additional threads on the host to manage the devices. To solve our synchronisation problem we start one thread for the host and one per device. For example, in a single device system, runs one host thread that handles the scheduling and host workload, and one device thread that handles the CUDA specific function calls for the device. This additional thread can be blocked by CUDA synchronizers and does not affect the host thread at all.

Another basic topic when programming with CUDA is memory management in heterogeneous systems. An NVIDIA GPU has a specific memory hierarchy as illustrated in Figure 2.3. On the lowest level, each thread has its local memory. On the next level, shared memory is available for all threads in one block. The highest level is global memory which is available to every thread and block of the device. We can allocate global device memory via the method *cudaMalloc()* and free it via *cudaFree()*. Host memory can be allocated in standard C fashion via the method *malloc()* and freed via *free()*. Also, it is possible to allocate pinned memory on the host via *cudaHostAlloc()*. Pinned memory stays always on the system memory and cannot be outsourced to storage memory. Further, it can be declared to mapped memory by using the flag "*cudaHostMapped*". This enables devices with compute capability greater than 1.0 to read and write directly on the data.

So the devices do not operate on their global memory but the system memory. Data can be directly accessed without first transferring it to the devices global memory. The two advantages of this method are that we can work on data that would not fit on the devices global memory which is typically smaller than the system memory and that we do not have to care about manually transferring data. All data transfer gets implicitly performed by the device when needed.

This leads us directly to the last topic of this introduction - data transfer and streams. The easiest way to manually copy data to a device is to use the method *cudaMemcpy()* with the "*cudaMemcpyHostToDevice*" flag. Vice versa when copying data back to the host with the flag "*cudaMemcpyDeviceToHost*". However, this copy mechanic has a downside - it is synchronous. Only after all the data was copied to the device the CUDA-kernel is called. This leads to a performance loss because modern GPUs have separated copy engines that can handle data transfer while kernels are running. To improve the performance, the idea is to copy a small partition of the assigned workload to the device and execute it thereafter, while the copy engine already copies the next partition. CUDA realizes this mechanic with the help of streams. To use streams the device must be able to handle overlap which supports nearly every NVIDIA GPU from compute capability 1.1 and above. In a stream, queued copy and kernel calls are executed sequentially while multiple streams are asynchronous among each other. So the idea is to create multiple streams, and overlap the data transfer with kernel calls of small workload partitions.

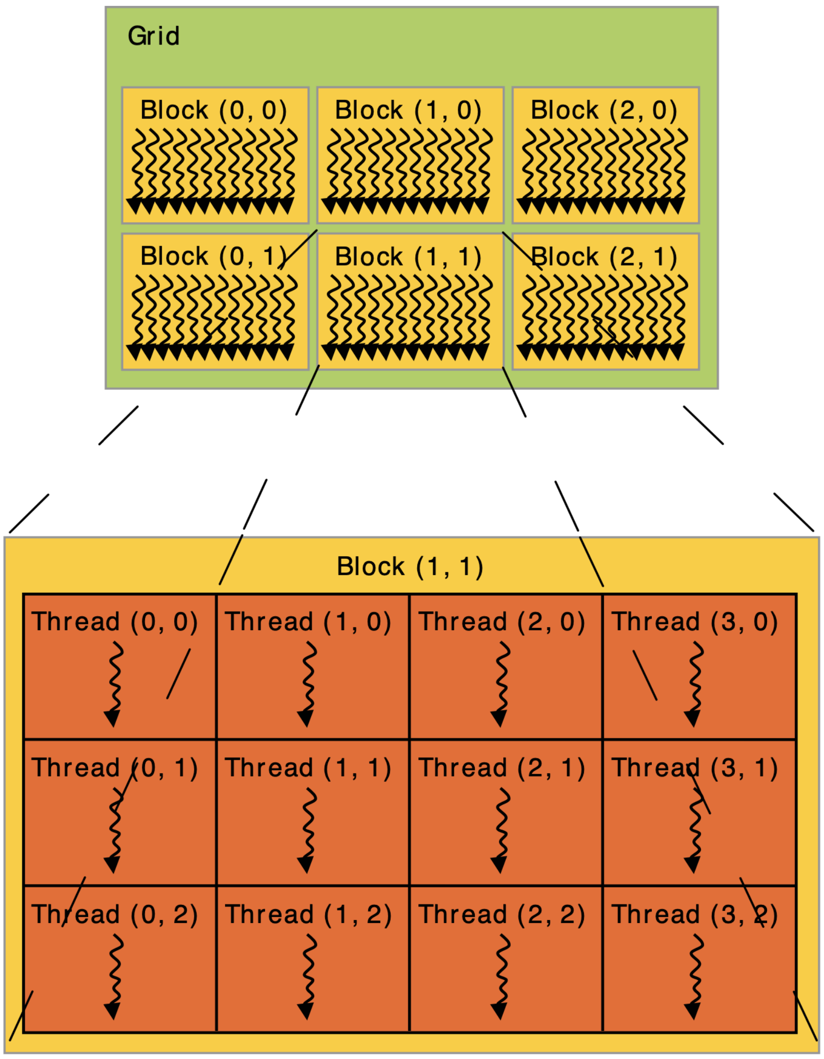


Figure 2.2: Scheme of the thread hierarchy on NVIDIA GPUs. The thread hierarchy consists of three levels. The highest level is the computation grid. The size of a grid is fixed by the second parameter of the kernel call. It contains an up to three-dimensional array of blocks. Each of these blocks accommodates a fixed number of threads. The thread is the lowest instance and executes the kernel code. The number of threads per block can be specified via the first parameter at the kernel call. Again, it is possible to use multidimensional structures. Furthermore, individual blocks can be accessed via *blockIdx* and the unique threads therein via *threadIdx* [Nvia].

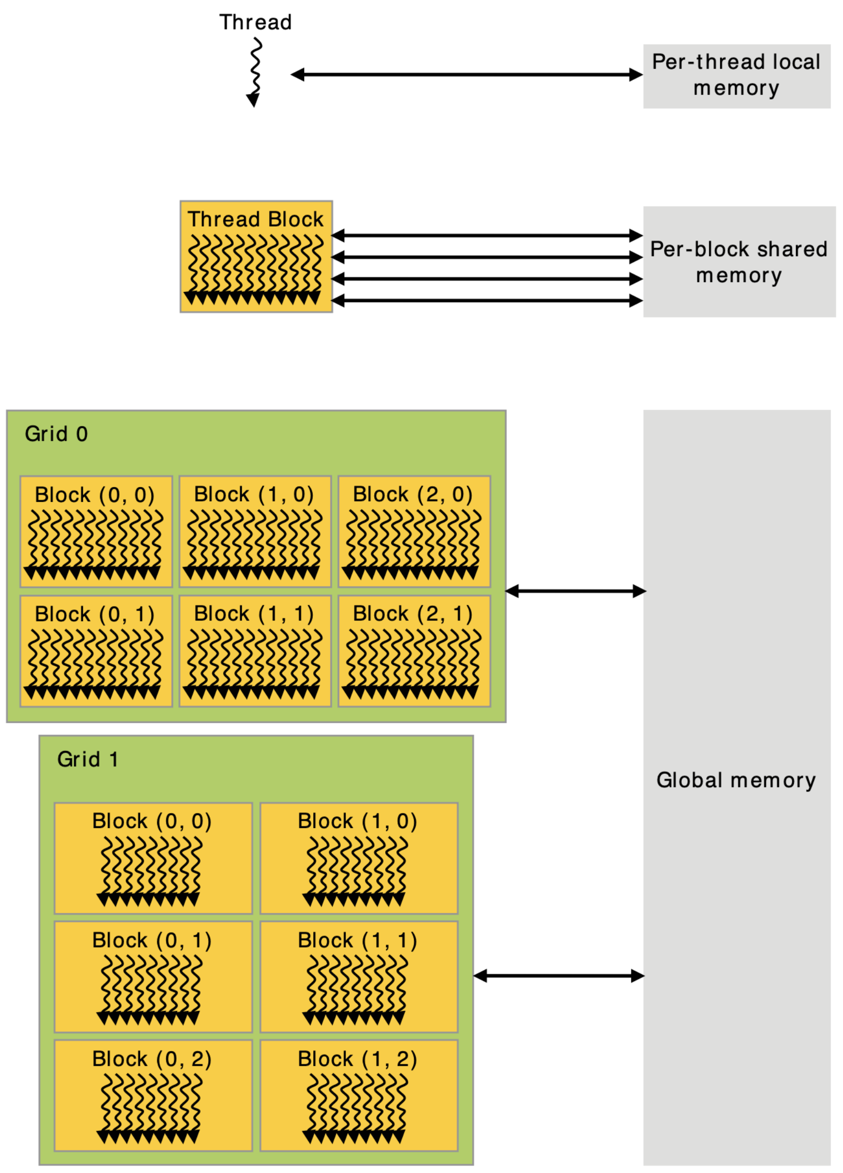


Figure 2.3: Scheme of the memory hierarchy on NVIDIA GPUs. The memory hierarchy works in hand with the thread hierarchy (see Figure 2.2). On the lowest level, each thread has its local memory. Then on the next level, there is shared memory per block to which all threads of a block have read and write access. The last level is the global memory where each thread in each block in each grid has access authorization [Nvia].

CHAPTER 3

Implementation

While the previous two chapters were part of the propadeuticum this chapter marks the beginning of the bachelor thesis. In this chapter, we first show the implementation of the kernels we use as workload. Then, we introduce our general performance estimation approach and derive multiple schedulers. In the end, we refer to other, more complex estimation approaches.

3.1 Kernels

Kernels are a fundamental part of scientific computations. We chose the two well-know level one BLAS-routines AXPY and DOT with different data-parallelism but the same memory-boundedness. Furthermore, we complement the BLAS-routines with a special kernel that computes the exponential function.

3.1.1 BLAS-kernels

In this subsection, we discuss the implementation of the two BLAS-routines AXPY and DOT. Scientific computations heavily rely on different levels of BLAS-routines reaching from simple vector addition (AXPY) up to matrix-matrix multiplication (GEMM) or the fast Fourier transform (FFT).

AXPY

The AXPY-routine may be one of the simplest BLAS-routines to implement for host and device. It is just an addition of a scaled vector **x** with another vector **y**. The algebraic equation is given by

(3.1) $\mathbf{z} = a \cdot \mathbf{x} + \mathbf{y}$

with the scalar a and where all three vectors have the same size and are real-valued. The host kernel is simply a for-loop over all assigned indices. It's implementation as CUDA-kernel is also straightforward. Both can be viewed in Algorithm 3.1.

Algorithm 3.1 AXPY-kernel

Implementation of the AXPY-kernel. For the host the kernel only consists of a for-loop which is iterating over the assigned workload. The assigned workload is defined by the start-index and end-index. The CUDA-kernel is almost the same, but each thread of a block works on its own partition until the overall index tid is bigger than the end-index.

```
//host kernel
void AXPY_host(int startIndex, int endIndex, float a, float* x, float* y, float* z)
{
  for (int i = startIndex; i < endIndex; i++)
  {
    z[i] = a * x[i] + y[i];
  }
}
//device kernel
__global__ void AXPY_device(int startIndex, int endIndex, float a, float* x, float* y, float* z)
{
  int tid = threadIdx.x + blockIdx.x * blockDim.x + startIndex;
  while (tid < endIndex)
  {
    z[tid] = a * x[tid] + y[tid];
    tid += blockDim.x * gridDim.x;
  }
}
```

Each thread per block solves the equation for its assigned indices until it reaches the end index. The best feature about the AXPY-routine is not the low coding complexity but its data-parallelism. The routine is embarrassingly data-parallel. This means each sub-equation at index i can be computed independently from the other sub-equations. We can simply cut off at any index and send the data to one device and the rest to another one and the result will be still correct. Therefore, we have a parallel computation complexity of $\mathcal{O}(N)$.
Further investigating the routine, we see that each sub equation consists of three load (a, x_i, y_i), two compute$(+, \cdot)$, and one store (z_i) instruction. This leads to the assumption that our results will be more dependent on the memory bandwidth than the raw single-precision performance of the system components. The memory-bound properties can be modelled by the naive roofline model [Wil08]. The component performance P can be model by

$$P = \min \begin{pmatrix} b \cdot I \\ f \end{pmatrix} \tag{3.2}$$

with peak performance f, bandwidth b and arithmetic intensity I. The arithmetic intensity is defined as the ratio between computation and communication work. For example, the arithmetic intensity of the SAXPY-routine is given by $I_{\text{SAXPY}} = \frac{1}{6}$ with computation cost $2N$ for the two compute instructions and communication cost $12N$ for two load and one store instruction.
As a result, the performance of an application is either capped by the peak performance or the memory bandwidth (see Figure 3.1).

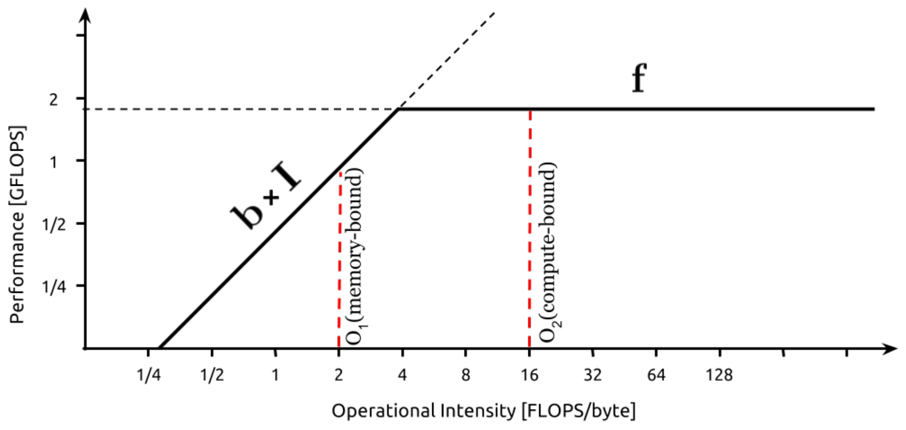


Figure 3.1: Naive roofline model. The performance of an application on a node is either capped by the bandwidth b or peak performance f. The greater the operational intensity of an application the more likely is it compute-bound. The intensity of O_1 is low such that its performance is memory-bound. In contrast, the intensity of O_2 is higher. Thus, its performance is compute-bound. Modified from [Nat16].

DOT

The second BLAS-routine we use is the DOT-routine. This level one BLAS-routine is just the scalar product of two vectors. The algebraic equation is given by

(3.3) $z = \mathbf{x}^T \cdot \mathbf{y}$

with the real-valued vectors **x** and **y** and the scalar result z. In contrast to the AXPY-routine, this routine is not embarrassingly data-parallel. Each sub-equation adds up to the result z. This creates a small overhead because after the host and all devices have finished the computation of their partial result, the host needs to add up all partial results. However, compared to the size of the workload this overhead is negligible. The data dependency also directly affects the CUDA-kernel because each block has to compute its partial result. Therefore, all threads in one block add up the products into the shared cache memory. This array then gets reduced via fan-in to one result per block. Then, the result of each block gets transferred into a helping array on the device-specific host thread. So the kernel does return an array and not a scalar value. The final reduction step has to be done on the device-specific thread. Because of the reduction steps, we get a parallel computation complexity of $\mathcal{O}(N \cdot log(N))$. Algorithm 3.2 shows the CUDA-kernel that reduces the large array into a way smaller one which can be then post-processed on the CPU. Again, we investigate the sub-equations which consist of two load (x_i, y_i), one compute$(\cdot)$, and one store $(cache_i)$ instruction. So the kernel is also memory-bound. But we may see a different performance compared to the AXPY-kernel because we have to copy fewer data back to the host.

Because both BLAS-kernels have memory-bound properties, we derive a compute-bound kernel in the next section.

3.1.2 Exponential kernel

Because level one and even level two BLAS-routines are memory-bound on GPUs [VD08], we derive a compute-bound kernel in this section. We expect for this kernel that our schedulers distribute the workload more in favour of the devices compared to the BLAS-routines. As workload we choose the computation of the exponential function, which is defined as

$$(3.4)\quad exp(x) := \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

with the scalar x. The sum up to its N-th part can be easily computed in $\mathcal{O}(N)$ by using $x^k = x^{k-1} \cdot x$ and $k! = (k-1)! \cdot k$ (see Algorithm 3.3). However, we choose a more naive approach and compute x^k and $k!$ each time from scratch (see Algorithm 3.4). This results in worse $\mathcal{O}(N^2)$ complexity. Because the EXP-kernel has to send back the reduced result of the sum we can use the same fan-in as for the DOT-kernel (see Algorithm 3.2) and post-process on the device thread.
Note that from a scientific point of view even an efficient computation of $exp(x)$ up to the N-th partial sum is useless. The contributions for large N get very small and vanish because we use floating-point accuracy. Nonetheless, there are two reasons to consider the naive EXP-workload. First, the EXP-kernel requires minimal data transfer (scalar x and the partial result of each block). And second, its scaling is linear i.e. the load for $j = 0, ..., \frac{N}{2}$ is three times smaller than for $j = \frac{N+1}{2}, ..., N$.

Algorithm 3.2 DOT-kernel

Implementation of the DOT-kernel. While for the single-threaded host one for-loop is sufficient to compute the partial result of the dot product, the CUDA-kernel faces the problem of data-parallelism. To compute the devices partial result we need to make multiple reduction steps and reduce the result of each thread to one per block. This reduction is done by using fan-in, where the number of adding threads is divided by two each cycle. In the end, we get an array which entries correspond to the partial result of each block. To get the final partial result of the device we need to make a final reduction step, that is not part of the CUDA-kernel.

```
//host kernel
float DOT_host(int startIndex, int endIndex, float* x, float* y)
{
  float sum = 0.0f;
  for (int i = startIndex; i < endIndex; i++)
  {
    sum += x[i] * y[i];
  }
  return sum;
}
//device kernel
__global__ void DOT_device(int startIndex, int endIndex, float* x, float* y, float* sumBlock)
{
  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x + startIndex;
  int cacheIndex = threadIdx.x;
  float temp = 0;
  while (tid < endIndex)
  {
    temp += x[tid] * y[tid];
    tid += blockDim.x * gridDim.x;
  }
  // set the cache values
  cache[cacheIndex] = temp;
  // synchronize threads in this block
  __syncthreads();
  // for reductions, threadsPerBlock must be a power of two
  int i = blockDim.x / 2;
  while (i != 0)
  {
    if (cacheIndex < i)
      cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
  }
  if (cacheIndex == 0)
  {
    sumBlock[blockIdx.x] = cache[0];
  }
}
```

Algorithm 3.3 Efficient $exp(x)$ computation

Computation of $exp(x)$ up to the N-th partial sum in $\mathcal{O}(N)$. Helping variables are used to store the current faculty and power of x.

```
float $sum = 1.0f$;
float $fac = 1.0f$;
float $x\_pot = x$;
for(int $k = 1; k < N + 1; k{+}{+}$)
{
  $fac = fac \ / \ k$;
  $x\_pot = x\_pot * x$;
  $sum = sum + fac * x\_pot$
}

```

Algorithm 3.4 Naive $exp(x)$ computation

Computation of $exp(x)$ up to the N-th partial sum in $\mathcal{O}(N^2)$. Instead of storing the intermediate results, we compute the faculty and power in each iteration from scratch.

```
float $sum = 1.0f$;
for(int $k = 1; k < N + 1; k{+}{+}$)
{
  float $fac = 1.0f$;
  float $x\_pot = x$;
  for(int $i = 1; k < N + 1; i{+}{+}$)
  {
    $fac = fac \ / \ i$;
    $x\_pot = x\_pot * x$;
  }
  $sum = sum + fac * x\_pot$
}
```

3.2 Scheduling algorithms

The essential core of each load balancing strategy is the scheduler. It uses information about the system to make a scheduling decision and assigns different workload partitions to the nodes of the system. Our iterative simulation scenario has to be optimized in two different variables:

- Minimize the workload computation time on each node through better and more costly scheduling
- Minimize the scheduling overhead which risks a worse scheduling decision that results in slower workload computation

These two variables can be combined into one criterion which minimizes the total computation time of each iteration. One iteration contains the scheduling decision of the host and the computation of the assigned workload on each node. There exist many different scheduling approaches but we use schedulers that have relatively low complexity and all use the same principle. The basis of our scheduling algorithms is the system performance p_{total} which is defined as

$$\begin{aligned} p_{total} &= \sum_{i}^{nodes} p_i, \\ p_{total} &= 1. \end{aligned} \tag{3.5}$$

Each node has a performance index (PI) p_i between 0 and 1, such that summed up the total system performance is 1. Let's assume our system only consist of the host and one device with $p_{host} = \frac{1}{4}$ and $p_{device} = \frac{3}{4}$. Then the scheduler assigns $\frac{3}{4}$ of the total workload to the device and the remaining $\frac{1}{4}$ to the host.

3.2.1 Static scheduling

As described in Chapter 2, static scheduling is dependent on the available information before the actual application run-time. Furthermore, a static scheduler does not create any scheduling overhead during the computation. We use a static scheduler as a reference and compare it against the dynamic schedulers derived in this work. The performance of the static scheduler is only dependent on our "initial guess" on how we estimate the system performance and initialize the PIs of the host and the devices. Algorithm 3.5 shows how the static scheduler assigns the workload by specifying the start and end index by multiplying the PI with the workload size.

No information If no information about the system is available before the run-time starts, a naive approach is to initialize all computing nodes equally [Lam15]. The PIs of the host and all devices are set to

$$p_i = \frac{1}{n} \tag{3.6}$$

with n being the number of nodes in the system. The bottleneck of this initial guess is the slowest part of the system. For example, the host could significantly slow down the iteration because the devices exploit their superior bandwidth. In a homogeneous system, however, this is the optimal solution if we assume that each node has no additional load.

Algorithm 3.5 Static scheduler

Static PI-based scheduler. The method has to be called only one time before the first iteration. It assigns each node a partition of the total workload corresponding to its PI-value. The size of the assigned workload is defined by the start-index and the end-index e.g. of the dot product vectors.

```
void staticScheduler (HostStruct* host, DeviceStruct* devices)
{
  int remainingWorkload = 0;
  //DISTRIBUTE WORKLOAD
  //Compute device workload
  for (int i = 0; i < deviceCount; i++)
  {
    //Assign device i workload
    devices[i].workloadDevice = (int)(devices[i].performanceIndexDevice * taskSize);
    //Map data to device
    devices[i].startIndex = remainingWorkload;
    remainingWorkload += devices[i].workloadDevice;
    devices[i].endIndex = remainingWorkload;
  }
  //Compute host workload
  host->startIndexHost = remainingWorkload;
  host->endIndexHost = taskSize;
  host->workloadHost = taskSize - remainingWorkload;
}
```

Type information If a bit more is known about the system, e.g. that all devices have the same specifications, we can manually specify a performance relation between the devices and the host. Let this relation be $r_{host} : r_{device}$, then we can initialize the PIs for the static scheduling as

$$\begin{aligned} p_{host} &= \frac{r_{host}}{r_{host} + n_{device} \cdot r_{device}}, \\ p_i &= \frac{r_{device}}{r_{host} + n_{device} \cdot r_{device}} \end{aligned} \tag{3.7}$$

for all devices i with the number of devices n_{device}. For example, $r_{host} : r_{device}$ could be fixed to 1:2 which means the devices are twice as fast as the host. In a system with one host and two devices, the PIs would be initialized as $\frac{1}{5}$ for the host and $\frac{2}{5}$ for each device. This initial guess is efficient and performs better than equal initialisation when the relation is chosen reasonably by the programmer and the set of devices is homogeneous. Big performance differences across the devices can act as a bottleneck.

Specification information The more information we have about the system the better can we approximate the performance of individual nodes. If we have access to detailed system information like memory bandwidth, clock speed, and core count of each node we can make a better estimation of the node's real performance and therefore make a better initial guess. Assume we have access to all the system information, then we can make a pretty good initial guess. That's why static scheduling approaches use different techniques to gather as much information as possible before the application is launched.

Semi-static The last static scheduling approach we talk about is not permanently static. It estimates the PIs by using the information gathered during the first iteration. The scheduler uses the initial PI distribution to measure the execution time of the assigned workloads for each node. Then before the second iteration, it uses this gathered information to estimate the PIs. In contrast to a fully dynamic approach, the PIs are constant for all following iterations. An advantage of the semi-static scheduler is that it needs no explicit information about the system and creates only a small scheduling overhead before the second iteration.
The main problem even an optimal static scheduler cannot overcome is its inability to react to sudden short or long performance changes of nodes that may occur during the run-time. Only dynamic schedulers have that ability.

3.2.2 Just in time performance scheduling (JITP)

In the last subsection, we talked about static scheduling and concluded that even the best initial guess can lead to decreased performance over time. To further optimize the iteration computation time we can measure the current performance of the host and each device each iteration and then rearrange the workload based on that information.
First, we introduce a dynamic scheduler that reacts immediately to performance changes. Because of this property we name it "*Just in time performance scheduler*". We use the execution time of the workload assigned to the host and the devices measured by CUDA events (see Section 2.2). After each iteration k we store the execution time t_i^k of each component. At the beginning of the next iteration $k+1$ the JITP-scheduler uses this information to calculate the current performance estimation for each node. It assumes that the node performance is constant in relation to the workload size N.

The estimations are given by

$$(3.8)\quad \hat{p}_i^{k+1} = \frac{w_i^k}{t_i^k}$$

with the size of the assigned workload w_i^k during iteration k. To match the performance model (Equations (3.5)) the estimations have to be normed. So we compute the total estimated performance

$$(3.9)\quad \hat{p}_{total}^{k+1} = \sum_i \hat{p}_i^{k+1}$$

and then update the PIs with the normed estimations

$$(3.10)\quad p_i^{k+1} = \frac{\hat{p}_i^{k+1}}{\hat{p}_{total}^{k+1}}.$$

After this update phase the schedulers assigns the workload in the same way as the static scheduler by multiplying the PI with the total workload (see Algorithm 3.6). Due to performance fluctuations of the nodes, we expect the PIs to change slightly after each iteration. Furthermore, a bad initial guess should be corrected after a couple of iterations. Also, the scheduler should react immediately to decreased performance at affected nodes - even if it was unlikely in the past. At the next iteration, these nodes get a smaller partition of the workload. And that might be its greatest flaw: It uses no information about the performance in the past.

3.2.3 Weighted performance scheduling (WP)

Instead of just using the latest information about the workload execution time the *weighted performance scheduler* uses all information available since the run-time started. The base assumption is that we get a better average performance and more stability compared to the JITP-scheduler when we use the entire information. The WP-scheduler is quite identical to the forgetting JITP-scheduler. It also uses the execution time t_i^k and calculates the performance estimation (Equation (3.8)) after each iteration. But the update step is different compared to Equation (3.10). Instead of simply updating the PIs the WP-scheduler takes the old PIs into account by the weight ω. The resulting updating term is given by

$$(3.11)\quad p_i^{k+1} = \omega \cdot p_i^{k+1} + (1-\omega) \cdot \frac{\hat{p}_i^{k+1}}{\hat{p}_{total}^{k+1}}$$

with $\omega \in [0,1]$. The implementation (see Algorithm 3.7) is almost identical to the JITP-scheduler and uses the familiar assigning principle. By changing the weight we can change the behaviour of the scheduler. A rational approach is to value the new information more than the old because its more recent e.g. $\omega = \frac{1}{2}$.

Algorithm 3.6 JITP-scheduler

Dynamic scheduler based on the performance estimation of the latest iteration. The algorithm can be divided into two steps. In the first step, the scheduler uses the time each component needed to compute its workload partition in the latest iteration to update the PIs of the system. The updating process follows Equations (3.8-3.10). In the second step, the scheduler assigns the new workload partition to the components based on the updated PIs. In fact, the second step is identical to the static scheduler (see Algorithm 3.5).

```
void JITPScheduler(HostStruct* host, DeviceStruct* devices)
{
  float unnormedPerformanceIndex[1 + deviceCount]; //first entry : host - then devices
  float performanceSum = 0;
  int remainingWorkload = 0;
  //UPDATE PERFORMANCE INDEX
  //Compute host performance
  unnormedPerformanceIndex[0] = host->workloadHost / host->elapsedTimeHost;;
  //Compute devices performance
  for (int i = 0; i < deviceCount; i++)
  {
    unnormedPerformanceIndex[i + 1] = devices[i].workloadDevice / devices[i].elapsedTimeDevice;
  }
  //Sum of total Performance to norm performance by computing performanceIndex
  for (int i = 0; i < deviceCount + 1; i++)
  {
    performanceSum += unnormedPerformanceIndex[i];
  }
  //Update performance index on host
  host->performanceIndexHost = unnormedPerformanceIndex[0] / performanceSum;
  //Update performance index on devices
  for (int i = 0; i < deviceCount; i++)
  {
    devices [i ].performanceIndexDevice = unnormedPerformanceIndex[i + 1] / performanceSum;
  }
  //DISTRIBUTE WORKLOAD
  //Compute device workload
  for (int i = 0; i < deviceCount; i++)
  {
    ...
  }
  //Compute host workload
  ...
}
```

Algorithm 3.7 WP-scheduler

Dynamic scheduler based on weighting the old PI with the latest performance estimation. The algorithm can be divided into two steps. In the first step, the PIs get updated - unlike the JITP-scheduler (see Algorithm 3.6) - by weighting the old PI with the newly computed estimation (see Equation (3.11)). In the second step, the scheduler distributes the workload the same way as the static and JITP-scheduler.

```
void WPScheduler(HostStruct* host, DeviceStruct* devices , float weight)
{
   float unnormedPerformanceIndex[1 + deviceCount]; //first entry: host − then devices
   float performanceSum = 0;
   int remainingWorkload = 0;
   //UPDATE PERFORMANCE INDEX
   //Compute host performance
   unnormedPerformanceIndex[0] = host−>workloadHost / host−>elapsedTimeHost;
   //Compute devices performance
   for (int i = 0; i < deviceCount; i++)
   {
     unnormedPerformanceIndex[i + 1] = devices[i].workloadDevice / devices[i].elapsedTimeDevice;
   }
   //Sum of total Performance to norm performance by computing performanceIndex
   for (int i = 0; i < deviceCount + 1; i++)
   {
     performanceSum += unnormedPerformanceIndex[i];
   }
   //Update performance index on host by weightening
   host−>performanceIndexHost = memoryWeight * host−>performanceIndexHost + (1.0f − weight) * (
    unnormedPerformanceIndex[0] / performanceSum);
   //Update performance index on devices by weightening
   for (int i = 0; i < deviceCount; i++)
   {
     devices [i ].performanceIndexDevice = weight * devices [i ].performanceIndexDevice
     + (1.0f − weight) * (unnormedPerformanceIndex[i + 1] / performanceSum);
   }
   //DISTRIBUTE WORKLOAD
   //Compute device workload
   for (int i = 0; i < deviceCount; i++)
   {
     ...
   }
   //Compute host workload
   ...
}
```

Furthermore, the WP-scheduler contains three special cases:

- $\omega = 1$: This case has the same effect as static scheduling because all new estimations are weighted 0 and the initialized PIs never change.
- $\omega = 0$: This case has the same effect as the JITP-scheduler as the old PIs are weighted 0.
- $\omega = \frac{k}{k+1}$: Using this weighting in iteration $k+1$ guarantees that the estimation of each iteration is weighted equally.

At the end of this section, we formulate our expectations on how the WP-scheduler behaves for different weights. For $\omega \to 0$ we expect the WP-scheduler to behave like the JITP-scheduler to which we already formulated our expectations in the last section. In contrast, we expect for $\omega \to 1$ different behaviour. A bad initialization may take very long to be balanced and the scheduling should not feel responsive to performance fluctuations.

3.2.4 Other dynamic scheduling models

The PI-based scheduling model used in this thesis takes a very generalized approach at gathering system information, as it only uses the total execution time of assigned workloads on the nodes. However, there exist other more complex approaches to estimate node performance. One way to improve the estimation accuracy is to split the kernel execution time and the time required for data transfer. Because the PCIe bus delivers an asymmetric bandwidth the data transfer time has to be split [MIRS14], [GAGZ+20] into the transfer time from host to device $t_{h\to d}$ and $t_{d\to h}$ vice versa. The time measurement now consists of

$$t_{i,total} = t_{i,h\to d} + t_{i,kernel} + t_{i,d\to h} \tag{3.12}$$

such that we get three performance estimations

$$\begin{aligned} p_{i,h\to d} &= \frac{s_{i,h\to d}}{t_{i,h\to d}} \\ p_{i,kernel} &= \frac{w_i}{t_{i,kernel}} \\ p_{i,d\to h} &= \frac{s_{i,d\to h}}{t_{i,d\to h}} \end{aligned} \tag{3.13}$$

with s_i being the size of the transferred data dependant on the workload size w_i and the workload type. Thus, making a scheduling decision is more complicated as the amount of transferred data is dependant on the workload type.

Our scheduling model assumes a constant node performance to workload size relation e.g. doubling the workload size doubles the computation time. [DW03] improves the accuracy by estimating a piece-wise linear performance function. Each node has its own piece-wise linear function $\mathcal{P}_i : N \rightarrow [0, 1]$ which is modified with new measurements at each iteration. In contrast, [LR07] and [CLR11] use a non-linear continuous performance function. The performance function can be estimated while or before the actual application run-time. A scheduler can make test iterations with pre-determined PIs e.g. ten iterations with $p_{host} = 0.1, ..., 1$ and $p_{device} = 0.9, ..., 0$ to estimate the performance function or start from scratch and estimate the function on-the-fly. There are many more variations of dynamic scheduling approaches. For more information consider the related work paragraph in Chapter 1.

CHAPTER 4

Results

As mentioned in previous chapters, we focus on dynamic schedulers that redistribute the workload after each iteration. For our benchmarks, we used two systems. System 1 was used for basic testing and coding. Its hardware may not be cutting edge by today's standards but has the performance of a typical consumer PC (see Figure 4.1(a)). In contrast, System 2's hardware is more powerful and contains two different GPUs of the same generation (see Figure 4.1(b)).
In the first section of this chapter, we compare the different scheduling algorithms proposed in Section 3.2. Furthermore, we explore their behaviour for different workload types and for different types of memory management. All result in the first section are based on System 1. In the second section we present the results based on System 2.

4.1 Host and one device

In this section, we test the performance of the scheduling algorithms introduced in Section 3.2 for various types of memory management. We use System 1 (see Figure 4.1(a)) for benchmarking.
At first, we test the performance improvements of dynamic schedulers compared to their static pendants. Therefore, we assume all data is stored on the system memory and map the data to the device, such that we do not have to manually copy data to the device. Then, we explore the performance gap between host and device by assuming static device memory and compare it to our theoretical estimation. Next, we manually transfer all the required data to the device using synchronous and asynchronous copying. Furthermore, we show the behaviour of the schedulers for memory-bound but load-heavy kernels and compute-bound kernels.

4.1.1 Mapped memory

We already talked about host allocated mapped memory in our short introduction GPU programming (see Section 2.2). To recall how to allocate and initialize mapped memory see Algorithm 4.1.

	System 1
CPU	**Intel i5 2500K**
Cores/Threads	4/4
Clock-speed	3.3Ghz
FLOPS (single/multi)	40GFlops/150GFlops
Memory	16GB DDR3 1600
Bandwidth	21GB/s
TDP	95W
GPU	**Nvidia Geforce GTX 1050Ti**
CUDA cores	768
Clock-speed	1506Mhz
FLOPS	2123GFlops
Memory	4GB GDDR5
Bandwidth	112GB/s
PCIe	3.0x16 @15,754GB/s
TDP	75W

(a) System 1

	System 2
CPU	**Intel i7 8700K**
Cores/Threads	6/12
Clock-speed	4.6Ghz
FLOPS (single/multi)	120GFlops/475GFlops
Memory	16GB DDR4 2666
Bandwidth	41,6GB/s
TDP	95W
GPU0	**Nvidia Geforce GTX 1080Ti**
CUDA cores	3584
Clock-speed	1607Mhz
FLOPS	11340GFlops
Memory	11GB GDDR5
Bandwidth	484GB/s
PCIe	3.0x8 @7,877GB/s
TDP	250W
GPU1	**Nvidia Geforce GTX 1050Ti**
CUDA cores	768
Clock-speed	1506Mhz
FLOPS	2123GFlops
Memory	4GB GDDR5
Bandwidth	112GB/s
PCIe	3.0x8 @7,877GB/s
TDP	75W

(b) System 2

Figure 4.1: Specifications[a] from [Int] and [Nvic] of the systems used for benchmarking. (a) System 1 consists of one Sandy-Bridge CPU and one Pascal GPU. (b) System 2 consists of one Coffee-Lake CPU and two Pascal GPUs.

[a]CPU FLOPS are taken from SGEMM Geekbench 4 score

For mapped memory, we also could use the "*cudaHostAllocWriteCombined*" flag. This flag enhances the performance for buffers that are read-only on the device [SK11]. However, it is useless for our scenario because it massively slows down CPU reads from the buffer. The workload consists of the kernels introduced in Chapter 3. We set the workload size N to the biggest power of two possible on System 1 which is $N = 2^{29}$ for mapped memory. This limit is given by

$$m_{required} = n_{vector} \cdot sizeOf(vector) = 3 \cdot 2^{29} \cdot 4Byte \approx 6.4GB \tag{4.1}$$

using float vectors. The following figures are all based on the AXPY-workload. In Figure 4.2 we see how the different scheduling types impact the iteration computation time.

Algorithm 4.1 Mapped memory

Code for allocating mapped memory. Instead of copying the data manually to the device, we simply hand over pointers. The data transfer is done automatically.

```
float* x, * dev_x;
//Allocate mapped memory on host
x = cudaHostAlloc((void**)&x, taskSize * sizeof(*x), cudaHostAllocMapped));
//Get device pointer to mapped memory
cudaHostGetDevicePointer(&dev_x, x, 0);
//Call kernel
kernel<<<1,1>>>(dev_x);
//Free memory
cudaHostFree(x);
```

All schedulers were initialized with one quarter of the total workload assigned to the host and three quarters to the device. The static scheduler (black) has the worst performance. This is obvious since the initial guess is non-optimal and does not change during the run-time. However, the computation time is not constant due to performance fluctuations of the system. A semi-static scheduling approach (green) yields better results. After the first iteration, the system performance is estimated well-enough to lead to a significant performance improvement. But this approach is a two-edged blade. The performance could be estimated very well after the first iteration, then the semi-static scheduler is as good as dynamic approaches. However, if the first PI estimation bad there is no room for further improvement.
The best results are provided by dynamic scheduling approaches. Comparing the JITP-scheduler (blue) to the WP-scheduler (red) with weight $\omega = 0.5$ we see that after ten iterations the performance difference is in the margin of fluctuations. The WP-scheduler performs worse until the optimum because it remembers the non-optimal initialization. So questions arise if additionally using the old PIs is meaningless and how the WP-scheduler behaves for certain weights. We discuss them later.

First, we take a look at the PI distribution of each iteration and at how many iterations the schedulers need to reach the optimal distribution (see Figure 4.3). As defined the static scheduler never changes the initial guess and the PIs are constant throughout the run-time. The semi-static scheduler behaves almost the same. After the first PI estimation, the distribution does not change again. Interestingly, the two dynamic schedulers behave quite equivalent. After five to ten iterations, the PIs do not change significantly. The WP-scheduler needs a few iterations more to reach the optimum. Assuming there is no other big load on the host or device, it might be the best idea to extend the estimation range of a semi-static scheduler (ESS) to e.g. ten iterations. Further, it might be interesting to see how the different scheduling approaches affect the workload execution time on the host and the device (see Figure 4.4). The device execution time is plotted in red while the host execution time is plotted in blue over the first ten iterations. Looking at the times of the static scheduler, we see that the initial guess overloaded the device quite heavily.

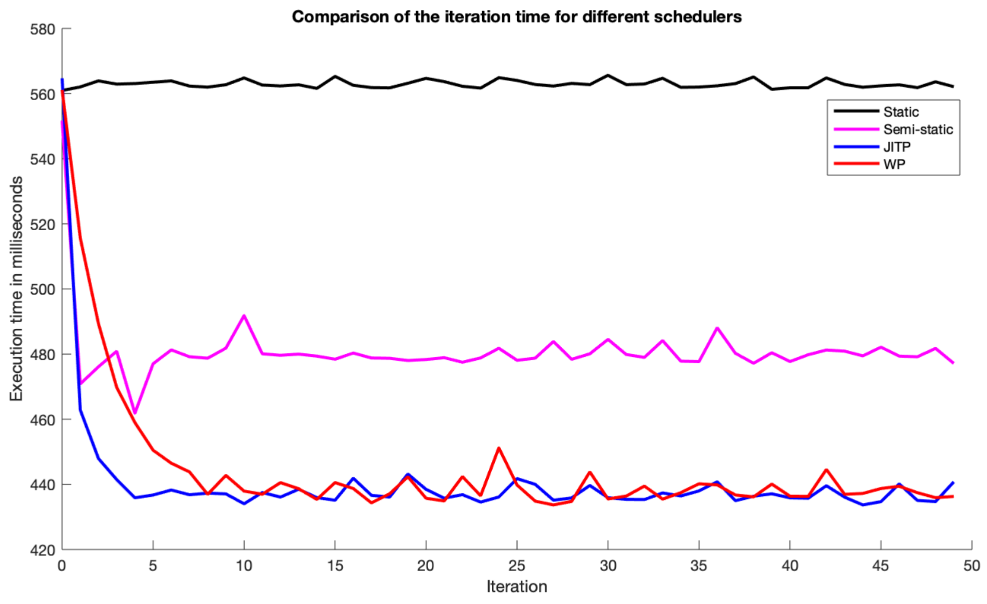


Figure 4.2: Comparison of the iteration computation for different schedulers. We timed 50 iterations where System 1 (see Figure 4.1(a)) computed an AXPY-workload of size 2^{29}. We used the schedulers from Chapter 3. The static scheduler (black) performs the worst followed by the semi-static approach (purple). The best results yield the dynamic JITP- (blue) and WP-scheduler (red).

So the other schedulers should redistribute the overload to the host. The semi-static scheduler cannot estimate the real performance well enough after the first iteration, so the device is still overloaded when it switches to static.
After only three iterations the JITP-scheduler has reached a state where host and device need the same time for executing their assigned workload. As seen in Figure 4.3, the WP-scheduler needs a few iterations more to reach the optimum compared to the non-weighting JITP-scheduler.
When we introduced the WP-scheduler we predicted that its behaviour depends on the weight ω. So we investigated the influence of the weight ω (see Figure 4.5). We conclude that for a memory weight ≤ 0.5 (blue, red) there is no big difference and the WP-scheduler behaves more like the JITP-scheduler. For high values e.g. 0.9 (purple), it takes longer to reach the optimum but the estimation is more resistant to performance fluctuations. In black, we additionally plotted the third special case where all iterations are weighted equally. At first, new values are weighted relatively high but after 20 iterations the new value is only weighted $\frac{1}{21}$ and has almost no impact at the scheduling decision.

In the next test, we investigate the long term behaviour of the JITP-, WP-, and the new ESS-scheduler. As proposed, the ESS-scheduler is a semi-static scheduler that uses the first ten iterations to estimate the system performance. Then, the PIs are static and the workload distribution does not change for the remaining run-time. For this test run, we put no other load at System 1. Figure 4.6 shows a table containing the iteration time statistics of a test run of 500 iterations. To neglect the influence of the initial guess we only used the data from the last 450 of 500 iterations. The WP-schedulers average performance is the best followed by the ESS- and JITP-scheduler. However, the performance improvement is marginal with less than a half per cent of the total iteration time. Due to that minor performance differences, we conclude that for an exclusively loaded system there is no real need for dynamic scheduling as semi-static approaches yield nearly the same results. However, it is interesting that the standard derivation of the WP-scheduler is smaller than the standard derivation of the other two algorithms. Weighting the previous PI with the latest estimation, may not be as unnecessary as the test results up to now suggested.

4.1.2 Static device memory

In contrast to mapped memory, we focus in this section on static device memory. The entire data stored on the devices global memory. There is no kind of data transfer to or from the device during the iterative computation. The intention of this scenario is to investigate how the greater memory bandwidth of the device in System 1 affects the workload distribution when we neglect data transfer. Before we take a look at the actual results, we make a theoretical estimation. As we assume the AXPY-kernel to be heavily memory-bound as it only performs two operations and has four memory accesses and neglect compute instructions, we expect the optimal workload distribution to be determined by the host and device bandwidth. The optimal PIs have to fulfil

$$\frac{1}{b_{host}} \cdot p_{host} = \frac{1}{b_{device}} \cdot p_{device} \tag{4.2}$$

with bandwidth b_{host} and b_{device} for host and device memory respectively. With the properties in Equations (3.5) we can solve for p_{host} and p_{device}. Using the data for our test system we get

$$\begin{aligned} p_{host} &= \frac{b_{host}}{b_{host} + b_{device}} &= \frac{21GB/s}{21GB/s + 112GB/s} &\approx 0.16, \\ p_{device} &= \frac{b_{device}}{b_{host} + b_{device}} &= \frac{112GB/s}{21GB/s + 112GB/s} &\approx 0.84. \end{aligned} \tag{4.3}$$

In Figure 4.7 we see how many iterations the JITP- and WP-schedulers ($\omega = 0.5$) need until they converge to the theoretical limit (4.3). Note that we have to shrink down the workload to 2^{28} to fit on the $4GB$ device memory. Furthermore, the host and device were initialized equally. With this initial guess, the schedulers can make a better first performance estimation because the JITP-scheduler has almost perfectly estimated the optimal distribution after the first iteration. Again the WP-scheduler needs a few iterations more to reach the optimal distribution. This optimal distribution is almost identical to our theoretical estimation which verifies the memory-boundedness of the AXPY-routine.

Up to now we only considered the memory-bound AXPY-kernel. However, our special EXP-kernel has different properties. It mainly contains local memory accesses and has a linear scaling over the workload size N. This means that our schedulers cannot properly estimate the performance because they assume a constant scaling over N. As the scaling is linear, the actual size of the workload up to index j is given by

$$\sum_{k=0}^{j} k = \frac{j \cdot (j+1)}{2} \tag{4.4}$$

with $0 \leq j \leq N$. To get the actual partition of the workload we use the ansatz

$$\alpha \cdot \frac{N \cdot (N+1)}{2} = \frac{j \cdot (j+1)}{2} \tag{4.5}$$

with $\alpha \in [0, 1]$ being the actual partition size. We can rewrite the ansatz to

$$\alpha = \frac{j \cdot (j+1)}{N \cdot (N+1)} \approx \left(\frac{j}{N}\right)^2 = p^2 \tag{4.6}$$

for large j, N. So we have to square the PIs to get the actual workload distribution. But first, we formulate an estimation based on the theoretical peak performance of the components. The optimal workload distribution is given by the solution of

$$\frac{1}{f_{host}} \cdot w_{host} = \frac{1}{f_{device}} \cdot w_{device} \tag{4.7}$$

with the peak performance f_{host} and f_{device} for host and device respectively. Using the data for our test system the solution is given by

$$\begin{aligned} w_{host} &= \frac{f_{host}}{f_{host} + f_{device}} &= \frac{40GFLOPS}{40GFLOPS + 2123GFLOPS} &\approx 0.0185, \\ w_{device} &= \frac{f_{device}}{f_{host} + f_{device}} &= \frac{2123GFLOPS}{40GFLOPS + 2123GFLOPS} &\approx 0.9815. \end{aligned} \tag{4.8}$$

So we expect that over 98 per cent of the actual workload is transferred to the device and less than two per cent remains on the host.

Figure 4.8 shows the PI distribution for the dynamic schedulers. Unlike for the memory-bound workloads, this time the first, smaller part of the workload remains on the host while the second, larger part gets assigned to the device. The optimal PIs are $p_{host} = 0.12$ and $p_{device} = 0.88$ which translates to

$$\begin{aligned} w_{host} &= p_{host}^2 &&= 0.0144, \\ w_{device} &= 1 - p_{host}^2 &&= 0.9856. \end{aligned} \tag{4.9}$$

The result almost exactly matches the theoretical estimation. However, it is more interesting how the different schedulers behave. The JITP-scheduler performs very well for constant scaling workloads. For the EXP-workload it struggles to accurately estimate the optimal PIs and oscillates. This is caused by the linear scaling of the workload as each redistribution is either under- or overestimated. In contrast to the JITP-scheduler, who eventually levels off, the ESS-scheduler may be stuck with a non-optimal distribution. The best result yields the WP-scheduler which balances out the oscillation by weighting the latest estimation with the old PI. It estimated the optimal PIs four times faster than the non-weighting approach. The result leads to another improvement of the semi-static approach where the PIs are estimated by a weighting-based scheduler. But then the burden of a bad initial guess has to be considered such that more iterations may be needed to estimate the optimal distribution.

4.1.3 Copy memory

So far we only considered scenarios were we just handed over pointers of mapped memory to the device or fixed the data on the devices global memory. Now, we transfer the assigned workload partition to the device each iteration. After computing the kernel the device transfers the result back to the host. In Chapter 2 we already talked about the CUDA functions that enable manual data transfer. Algorithm 4.2 shows how to copy and copy-back a vector. Thus, this method is synchronous because the device thread on the host has to wait until the copying is complete. Then, the device thread calls the kernel. So data transfer acts as bottleneck. We can improve the copy performance up to twice the speed, when using pinned memory [SK11]. But still, data transfer slows down the device and therefore lowers its PI.
Modern GPUs try to solve this problem by using separate copy engines that allow data transfer while computing a kernel. This mechanic can be implemented with the help of CUDA streams that use the separate engines effectively. The idea is to split the workload into smaller partitions and to overlap the copy and kernel calls. Streams are synchronous considering their copy and kernel calls, but asynchronous to other streams. [CVKG10] showed that asynchronous data transfer of non-zero streams is slower than synchronous transfer with the zero stream. The performance gain has to be achieved by efficiently overlapping copy and kernel calls. Furthermore, it showed that above a certain data size asynchronous transfer is faster than mapped memory.

Algorithm 4.2 Synchronous data transfer

Code for synchronous data transfer. Device memory has to be allocated manually. After synchronously copying the data to the devices global memory, the kernel can be launched. The results are written into the device memory. Therefore, they have to be synchronously copied back to the host.

```
float* x, * dev_x;
//Allocate memory on host
x = malloc((void**)&x, taskSize * sizeof(*x));
//Allocate memory on device
dev_x = cudaMalloc((void**) &dev_x, sizeof(float) * taskSize));
//Copy data to device
cudaMemcpy(dev_x, &x, sizeof(float) * taskSize, cudaMemcpyHostToDevice));
//Call kernel
kernel<<<1,1>>>(dev_x);
//Copy data back to host
cudaMemcpy(&x, dev_x, sizeof(float) * taskSize, cudaMemcpyDeviceToHost));
//Free memory
free(x);
cudaFree(dev_x);
```

A significant performance increase can be achieved by using two streams (see Figure 4.9). Note that when multiple streams are used it is necessary to specify the stream that calls the CUDA method (see Algorithm 4.3). One stream copies a data partition to the device. While it executes the kernel on the partition, another stream copies the next partition to the device. Then, while the second stream executes the kernel, the first stream copies the result back to the host. The result of the second stream can then be copied back and overlapped with the first stream copying the next partition to the device. And so on until all partitions have been computed. As we already analysed the behaviour of the different schedulers we now use the WP-scheduler with memory weight $\omega = 0.5$ and compare the PI distribution for synchronous and asynchronous data transfer.
But first, we take a look at the AXPY-routine which has to copy two vectors to the device and copy the resulting vector back to the host. The total amount of transferred data in iteration k is given by

$$data^k = 3 \cdot w^k_{device} \cdot 4Byte. \tag{4.10}$$

Also, we can formulate a theoretical expectation about the resulting workload distribution. We modify Equation (4.2) by assuming that all the data accessed also needs to pass the PCIe bus. The new optimal PIs are given by the solution of

$$\frac{1}{b_{host}} \cdot p_{host} = (\frac{1}{b_{device}} + \frac{1}{b_{bus}}) \cdot p_{device}. \tag{4.11}$$

Algorithm 4.3 Asynchronous data transfer

Code for asynchronous data transfer. The device memory has to be allocated manually. After creating two streams, we can iterate over small workload partitions and both streams asynchronously copy data to the device. The kernel launch and the back-copying of results is also handled by both streams. The stream which is responsible for the function call has to be stated.

```
float* x, *dev_x_0, * dev_x_1;
//Define partion size
int partition = taskSize / 256;
//Allocate pinned memory on host
x = cudaHostAlloc((void**)&x, taskSize * sizeof(*x), cudaHostAllocDefault));
//Allocate memory on device
cudaMalloc((void**)&dev_x_0, partition * sizeof(float));
cudaMalloc((void**)&dev_x_1, partition * sizeof(float));
//Create streams
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
//Do asynchronous copying and execute small kernel
for (int i = 0; i < taskSize − 1; i += 2 * partition)
{
  //Asynchronous copying to device
  cudaMemcpyAsync(dev_x_0, &x[i], partition * sizeof(float), cudaMemcpyHostToDevice, stream0);
  cudaMemcpyAsync(dev_x_1, &x[i + partition]), partition * sizeof(float), cudaMemcpyHostToDevice,
   stream1);
  //Call kernel
  kernel<<<1,1,stream0>>>(dev_x);
  kernel<<<1,1,stream1>>>(dev_x);
  //Asynchronous copying back to host
  cudaMemcpyAsync(&x[i], dev_x_0, partition * sizeof(float), cudaMemcpyDeviceToHost, stream0));
  cudaMemcpyAsync(&x[i], dev_x_1, partition * sizeof(float), cudaMemcpyDeviceToHost, stream1));
}
//Synchronize streams
cudaStreamSynchronize(stream0));
cudaStreamSynchronize(stream1));
//Free memory
cudaFreeHost(x);
cudaFree(dev_x_0);
cudaFree(dev_x_1));
//Destroy streams
cudaStreamDestroy(stream0);
cudaStreamDestroy(stream1);
```

With the condition $p_{host} + p_{device} = 1$ we can solve Equation (4.11) for p_{host} and p_{device}. Using the data for our test system the optimal PIs are given by

$$\begin{aligned} p_{host} &= \frac{b_{host} \cdot b_{device} + b_{host} \cdot b_{bus}}{b_{host} \cdot b_{device} + b_{host} \cdot b_{bus} + b_{device} \cdot b_{bus}} &\approx 0.60, \\ p_{device} &= \frac{b_{device} \cdot b_{bus}}{b_{host} \cdot b_{device} + b_{host} \cdot b_{bus} + b_{device} \cdot b_{bus}} &\approx 0.40. \end{aligned} \tag{4.12}$$

So at best, around 40 per cent of the total workload should get assigned to the device, depending on the PCIe bus usage. Therefore, we expect a performance drop for the synchronous version. The questions rising are how big this performance drop actually is and if an asynchronous approach can hide the communication cost. Figure 4.10 provides answers to those questions. We use the known - device favouring - initial guess and a workload size of 2^{28}. The PI distribution for the synchronous version is even worse than we predicted. Nearly 70 per cent of the total workload stays on the host while only around 30 per cent gets copied to the device. The difference between estimation and reality is likely caused by the data transfer engine of the device that cannot take advantage of the full bus bandwidth. For example, if we assume that the engine only utilizes around $\frac{2}{3}$ of the PCIe 3.0x16 bandwidth we get a more fitting estimation. The asynchronous version performs better and the distribution is almost even. But two streams cannot overcome the performance loss compared to static device memory (see Figure 4.7(b)). However, the results for asynchronously copying the data to the device are slightly better than using mapped memory (see Figure 4.3(d)) and letting the device transfer the data automatically.
In contrast to the AXPY-routine, the DOT-routine only has to copy the partial result of each block back to the host. The data transferred each iteration is given by

$$data^k = 2 \cdot w^k_{device} \cdot 4Byte + numberBlocks \cdot 4Byte \tag{4.13}$$

which simplifies to

$$data^k \approx 2 \cdot w^k_{device} \cdot 4Byte \tag{4.14}$$

because $w^k_{device} \gg numberBlocks$. This means the DOT-kernel transfers only $\frac{2}{3}$ of the data for the AXPY-kernel. We should receive a visible performance gain on the device especially for the synchronous version. However, the question is if the required post-processing on the device thread compensates the gain or maybe even worsens the performance. The PI distributions for the synchronous and asynchronous DOT-workload can be seen in Figure 4.11. For the synchronous version, the PI distribution is only slightly different from the AXPY distribution. The device gets a slightly larger partition of the workload. It seems like the post-processing on the device thread slows down the execution time to a degree where the theoretical performance gain compared to the AXPY-routine vanishes. For the asynchronous version, the performance of the device is even worse than for the AXPY-routine. This may be caused by our specific choice of implementing the DOT-routine in which only uses one block to compute a partition to reduce device thread overhead.

In the last test of this section, we investigate the PI distribution of the WP-scheduler when only one copy cycle is required for many kernel launches. This simulates a load-heavy but still memory-bound workload. We expect the data transfer to be negligible and results in the region of static device memory (see Figure 4.7(b)). In Figure 4.12(b) we see the PI distribution for an AXPY-kernel launched 100 times between a copy cycle. As expected, the optimal distribution is almost the same as in a non-copy scenario. This is caused by the fact that we only have to use the bus bottleneck one time and profit 100 times from the devices superior bandwidth. The distribution does not shift towards the device when we use mapped memory because the data is transferred automatically in each of the 100 kernel launches (see Figure 4.12(a)). Therefore, it is more efficient to manually copy data onto the device if the memory-bound kernel is load-heavy enough.
Furthermore, we conclude that the best performance is achieved when the data is stored on the component with the superior bandwidth. If an application requires communication, asynchronous copy instructions are the way to go. The more complicated code is well worth the effort and yields noticeable performance gains compared to synchronous data transfer. If the device has not enough space to store the data, it makes sense to use mapped memory.

4.2 Host and multiple devices

In this section, we present our test result on another system which consists of a high-end consumer CPU, high-end consumer GPU, and low-end consumer GPU (see Figure 4.1(b)). We refer to this heterogeneous system as System 2.

We analyse how the changes in component bandwidth affect the performance and if the results match our memory-bound estimations. For all benchmarks, the dynamic WP-scheduler with $\omega = 0.5$ was used.

4.2.1 Performance estimation

Again, we assume that all the data used during the kernel execution is stored on the devices and there is no data transfer required. For our memory-bound kernels, we can again formulate a performance estimation, depending on the theoretical peak bandwidth of the system components. Because System 2 has three components we have to solve

$$\frac{1}{b_{host}} \cdot p_{host} = \frac{1}{b_{device0}} \cdot p_{device0} = \frac{1}{b_{device1}} \cdot p_{device1}. \tag{4.15}$$

The optimal PIs for System 2 are given by

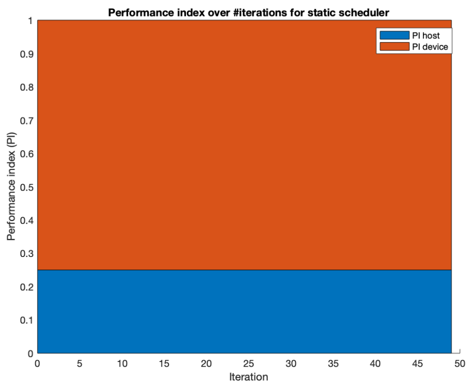
$$\begin{aligned} p_{host} &= \frac{b_{host}}{b_{host}+b_{device0}+b_{device1}} &\approx 0.06, \\ p_{device0} &= \frac{b_{device0}}{b_{host}+b_{device0}+b_{device1}} &\approx 0.76, \\ p_{device1} &= \frac{b_{device1}}{b_{host}+b_{device0}+b_{device1}} &\approx 0.18. \end{aligned} \tag{4.16}$$

Figure 4.13(a) shows the PI distribution for the memory-bound AXPY-workload. The three components are initialized equally and the WP-scheduler needs around five iterations to reach the optimal distribution. Device0 gets by far the biggest part of the workload. The two other components only get small parts. The result fits our estimation almost perfectly and differs only by a few per cent from the theoretical memory-bound optimum. Even though device0 has many more cores and a higher clock speed compared to device1, the performance difference is only caused by the superior bandwidth of device0. So, for memory-bound problems the only performance indicator is the device bandwidth.
As all components are initialized equally the host is overloaded and slows down the computation by factor five during the first iteration (see Figure 4.13(b)). The Figure shows how bad an initial guess can be if we do not use dynamic scheduling and that for efficient load balancing it is essential to somehow estimate the system performance before (static) or during (dynamic) the run-time. If we know the properties of the workload, e.g. AXPY is memory-bound, Equation (4.16) is a good and simple performance-based initial guess.

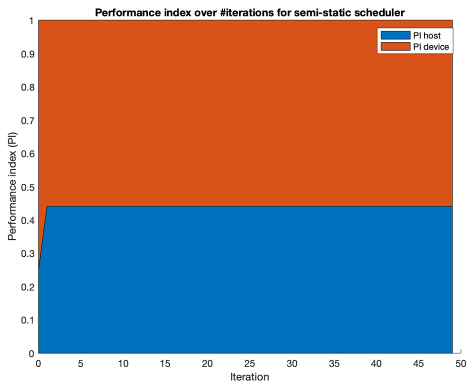
4.2.2 Data transfer comparison

We reviewed different ways of data transfer in Section 4.1. As there are no theoretical differences to a multi-device system we only share the results for System 2. Figure 4.14 shows the PI distributions and component execution times plotted for the two manual copy (synchronous/asynchronous) and the automatic copy (mapped memory) mechanisms. When we compare the PI distribution of synchronous transfer for System 1 (see Figure 4.10(a)) and System 2 (see Figure 4.14(a)) we see the same host-devices relation of around 60:40. However, the two devices split the 40 per cent equally. Even though device0 has a superior bandwidth compared to device1, they get the same partition because of the PCIe x8 bus bottleneck.
The asynchronous version yields better results than the synchronous (see Figure 4.14(c)). Again, both devices get the same partition of the workload. Comparing asynchronous data transfer and mapped memory (see Figure 4.14(e)), the PIs differ only by a few per cent and this time mapped memory yields slightly better results.

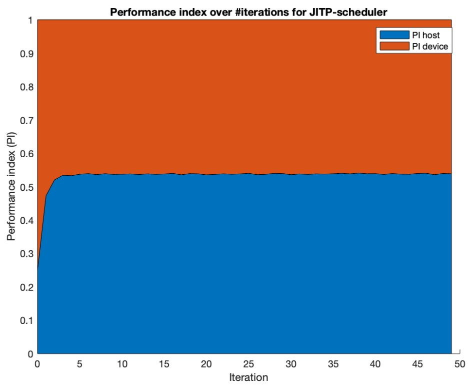
Moreover, we take a look at our iterated kernel scenario to simulate a memory-bound and load-heavy workload. Similar to System 1, we get expected results. For mapped memory (see Figure 4.15(a)) the performance matches the non-iterated kernel results (see Figure 4.14(e)) while for synchronous data transfer (see Figure 4.15(b)) the distribution is similar to the static device memory scenario (see Figure 4.13(a)).



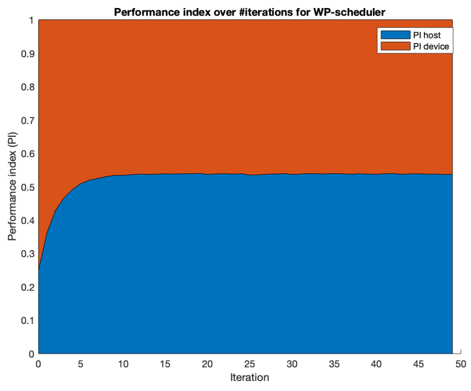
(a) Static scheduler



(b) Semi-static scheduler

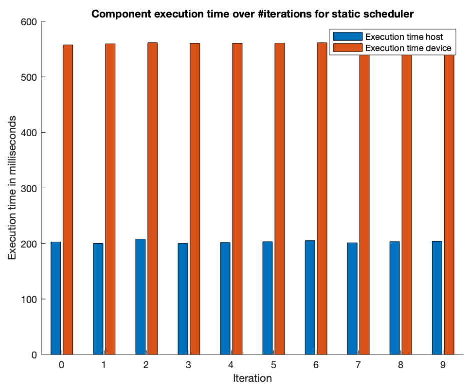


(c) JITP-scheduler

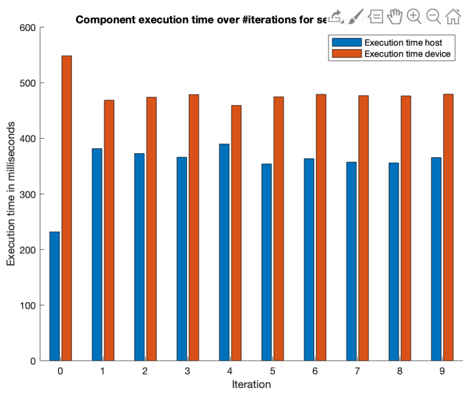


(d) WP-scheduler

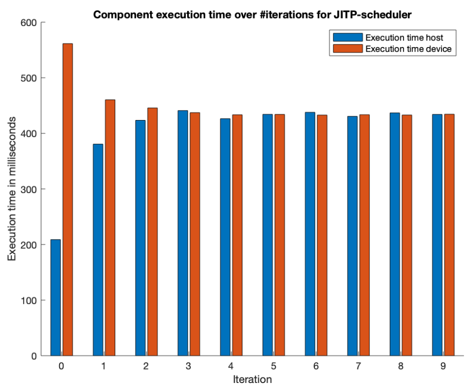
Figure 4.3: Comparison of the PI distribution for different scheduling approaches. We benchmarked System 1 (see Figure 4.1(a)) with an AXPY-workload of size 2^{29}. The blue area marks the PI of the host while the red shows the PI of the device. According to the PI, the schedulers assign the workloads. (a) Shows the distribution of the static scheduler. Obviously, it is constant. (b) Shows the distribution of a semi-static approach where after one estimation the PIs are constant. (c) Shows the distribution of the JITP-scheduler that only uses the latest estimation for decision making. After only a few iterations the PIs are well enough estimated for the distribution to be approximately constant. (d) Shows the distribution of the WP-scheduler with $\omega = 0.5$. Compared to the JITP variant it needs more iterations to estimate the optimal distribution.



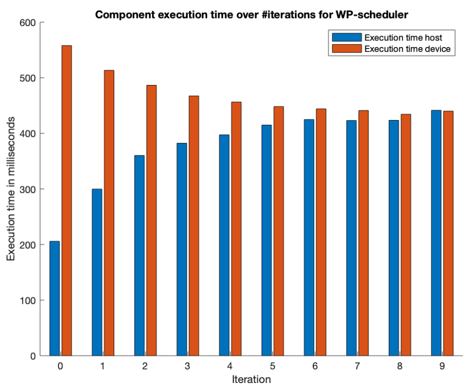
(a) Static scheduler



(b) Semi-static scheduler



(c) JITP-scheduler



(d) WP-scheduler

Figure 4.4: Comparison of the component execution time for different scheduling approaches. System 1 (see Figure 4.1(a)) computed an AXPY-workload of size 2^{29}. (a) Shows the comparison for the static scheduler. It becomes clear that the initial guess overloaded the device. (b) Shows the comparison for a semi-static scheduler. The first estimation is not good enough, such that the device is still overloaded. (c) Shows the comparison for the JITP-scheduler. After only three iterations the components are loaded equally. (d) Shows the comparison for the WP-scheduler. Because it remembers the bad initialization, the overload on the device gets redistributed slower.

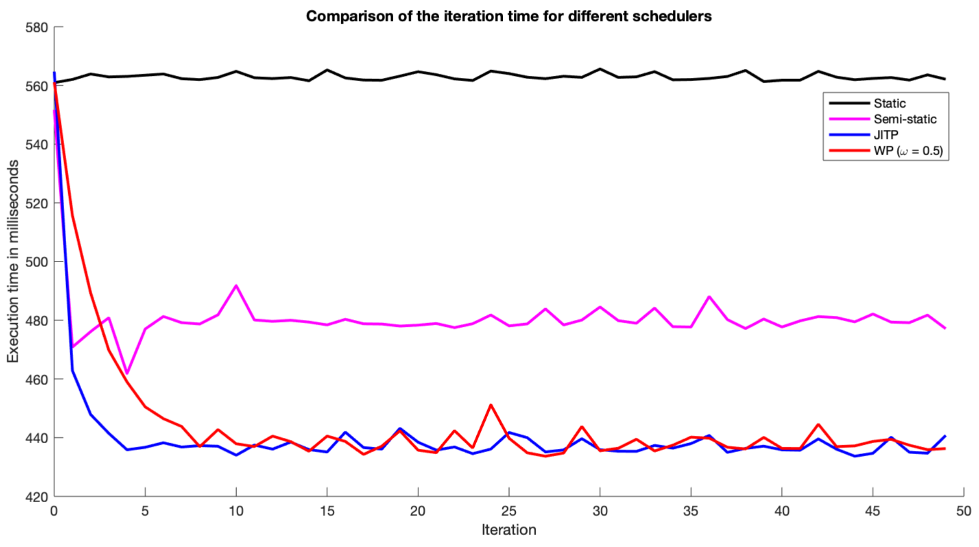
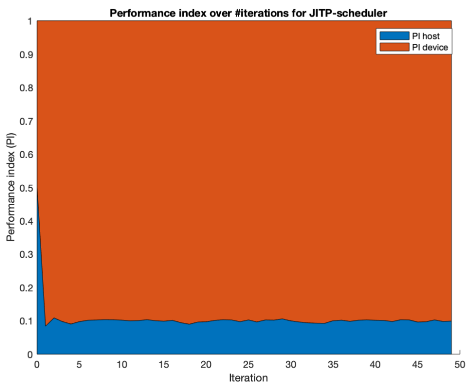


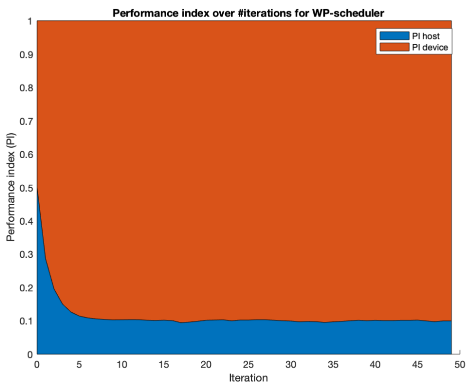
Figure 4.5: Influence of the weight ω on the behaviour of the WP-scheduler. The Figure shows the iteration computation time for different weights. The smaller the weight the more the WP-scheduler behaves like the JITP-scheduler. For bigger weights, it takes more iterations to reduce the iteration computation time. Weighting each value equally (black) proofs to be a bad choice because after a few dozen iterations $\omega \approx 1$ and new values have almost no influence.

	ESS-scheduler	JITP-scheduler	WP-scheduler
Min	437.79ms	437.77ms	436.42ms
Max	474.45ms	480.23ms	468.85ms
Mean	444.70ms	445.63ms	443.72ms
Variance	14.92ms²	18.40ms²	12.33ms²
Standard derivation	3.86ms	4.23ms	3.51ms

Figure 4.6: Iteration time statistics of different schedulers on System 1. As workload, we chose an AXPY-workload of size 2^{29}. The test consisted of 500 iterations, where only the data of the last 450 iterations was used for the statistics. The JITP- and WP-schedulers are already known from Chapter 3. The ESS is a semi-static scheduler that estimates the system performance during the first ten iterations. We see that the iteration times are in the range of $445 \pm 20ms$. The mean performance is almost equal. However, the WP-scheduler has a slight advantage. Furthermore, it has the lowest standard derivation. The JITP-scheduler performed worse than the ESS-scheduler.

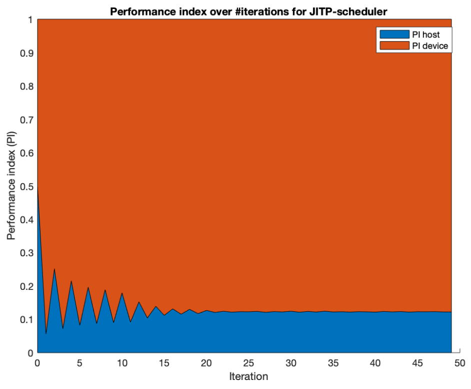


(a) JITP-scheduler

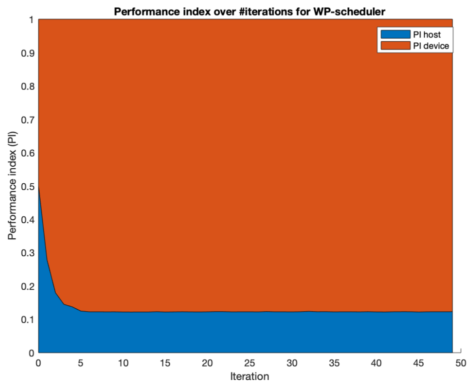


(b) WP-scheduler

Figure 4.7: PI distribution for the two dynamic schedulers with static device memory. Due to memory constraints on the device the AXPY-workload size was reduced to 2^{28}. Further, we used equal initialization. (a) Due to an equally distributed load, the JITP-scheduler nearly estimated the theoretical optimum after the first iteration. (b) The WP-scheduler needs more iterations to reach the optimum. Both distributions are quite close to the theoretical memory-bound optimum.



(a) JITP-scheduler



(b) WP-scheduler

Figure 4.8: PI distribution for the two dynamic schedulers with static device memory. A compute-bound EXP-workload of size 2^{14} was computed. (a) The JITP-scheduler oscillates and needs roughly 20 iterations to estimate the optimal PIs. (b) The WP-scheduler ($\omega = 0.5$) is more robust because it also considers the previous PIs and only needs five iterations to estimate the optimal PIs.

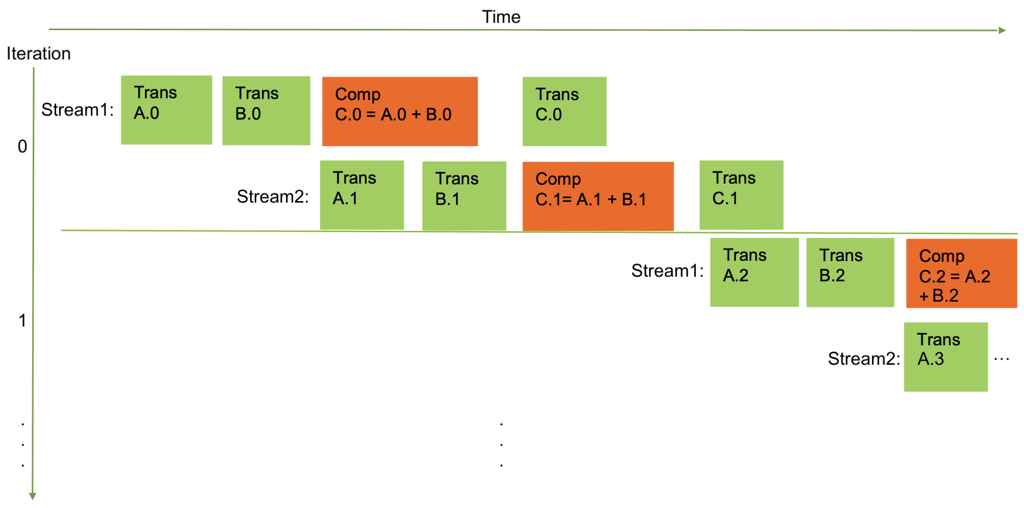
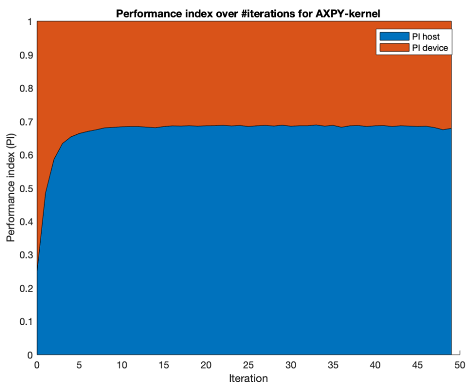
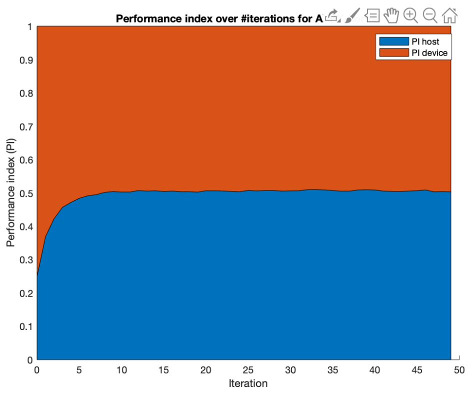


Figure 4.9: Stream overlap example using two streams. The green rectangles are copy calls e.g. "Trans A.0" copies partition 0 of vector A to the device. Moreover, the red rectangles are kernel calls e.g. "Comp C.0 = A.0 + B.0" computes the sum of vector A and B for partition 0. The idea is to use two streams and overlap the copy and kernel calls of adjacent partitions by assigning them alternately to the streams. During each iteration, both streams execute a copy-to-device, kernel and copy-to-host call. Modified from [HC].

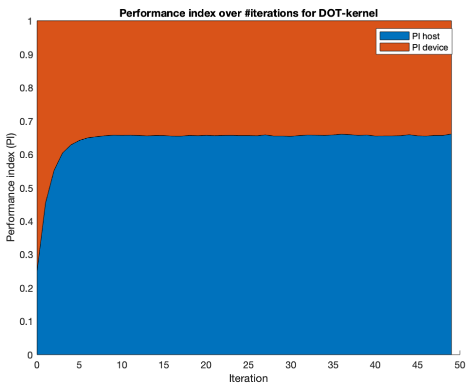


(a) Synchronous

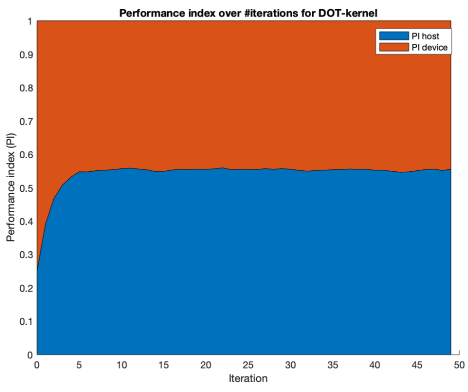


(b) Asynchronous

Figure 4.10: PI distribution for synchronous and asynchronous data transfer on System 1 (see Figure 4.1(a)) plotted for an AXPY-workload of size 2^{28}. (a) The optimal distribution for synchronous transfer has a 68-32 host-device ratio. So there is an eight per cent difference between the theoretical estimation (4.12) and the actual result. (b) For asynchronous transfer, the optimal distribution is 51-49. Therefore, the overall performance of asynchronous copying is slightly better than the performance of mapped memory.

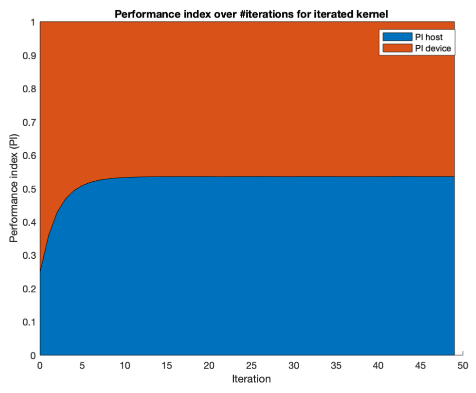


(a) Synchronous

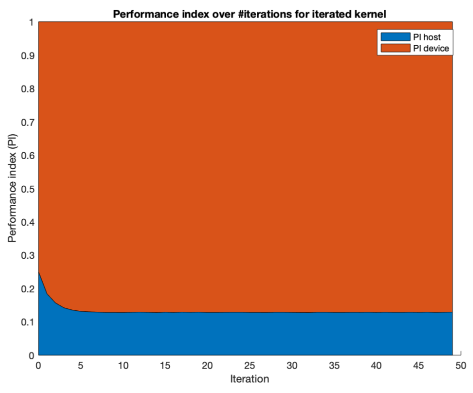


(b) Asynchronous

Figure 4.11: PI distribution for synchronous and asynchronous data transfer on System 1 (see Figure 4.1(a)) plotted for a DOT-workload of size 2^{28}. (a) For synchronous copying, the optimal distribution has a ratio of 66-34. Even though the DOT-routine transfers less data the distribution is nearly the same as the AXPY-workload distribution (see Figure 4.10(a)). (b) Compared to the AXPY-workload, asynchronous transfer yields a smaller performance gain. The device's performance is bottlenecked by the post-processing on the device thread.

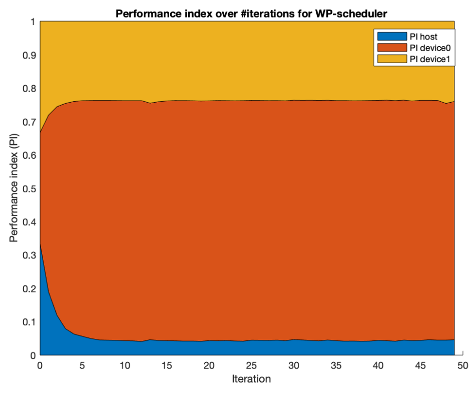


(a) Mapped memory

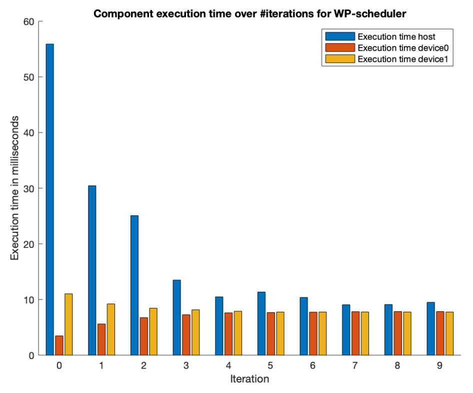


(b) Synchronous

Figure 4.12: PI distribution for an AXPY-kernel iterated 100 times between each data transfer cycle on System 1. The workload was initialized unequally and scheduled by the WP-scheduler with $\omega = 0.5$. (a) For mapped memory, we see no difference compared to one kernel launch (see Figure 4.3(d)). (b) For a synchronous copy cycle around the iterated kernel we see a PI distribution that matches the static device memory estimation (4.3). Thus, the data transfer time can be neglected.

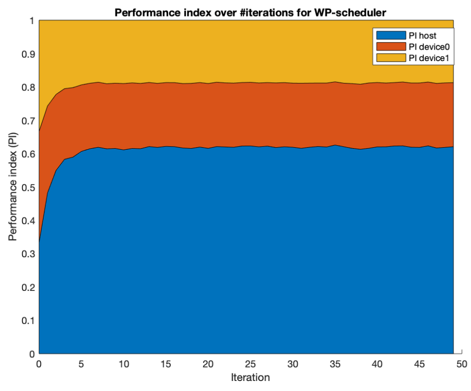


(a) PI estimation static device memory

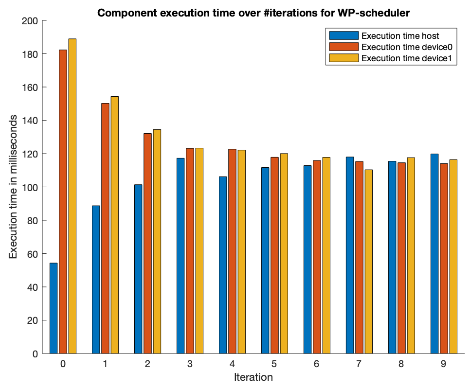


(b) Component execution time comparison static device memory

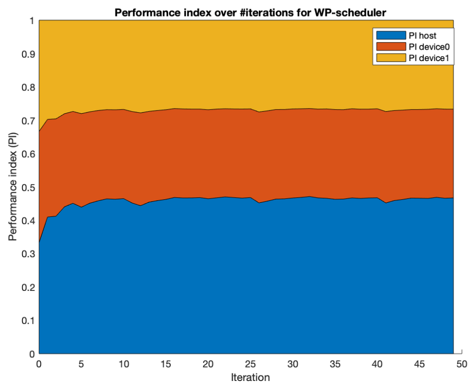
Figure 4.13: PI distribution and component execution time for static device memory on System 2 (see Figure4.1(b)). An AXPY-workload of size 2^{28} was scheduled over 50 iterations by the WP-scheduler with $\omega = 0.5$. (a) The PI distribution is approximately constant after five iterations and matches the theoretical estimations from Equations (4.16). At first, all components were initialized equally. But when the workload is optimally distributed, device0 computes over 75 per cent of the total workload. (b) The initial guess heavily overloads the host until the workload gets redistributed to device0.



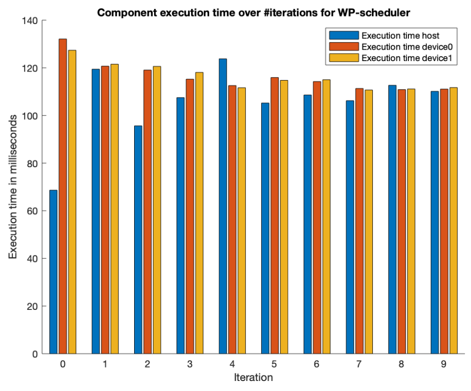
(a) PI distribution synchronous



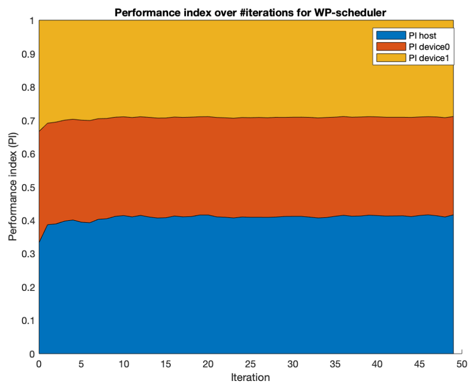
(b) Component execution time comparison synchronous



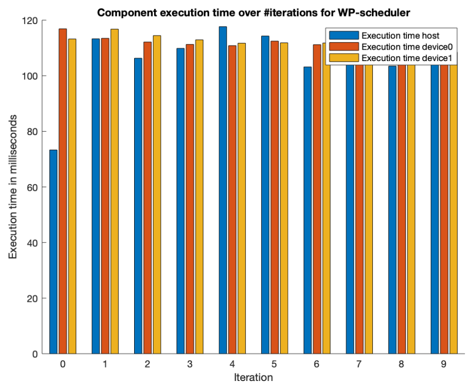
(c) PI distribution asynchronous



(d) Component execution time comparison asynchronous

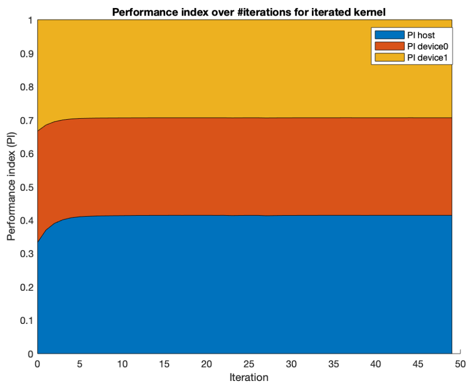


(e) PI distribution mapped memory

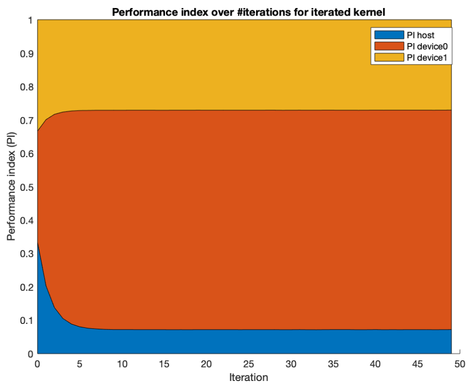


(f) Component execution time comparison mapped memory

Figure 4.14: PI distributions and component execution times for various data transfer scenarios on System 2 (see Figure 4.1(b)). An AXPY-workload of size 2^{28} was scheduled over 50 iterations by an WP-scheduler with $\omega = 0.5$.



(a) Mapped memory



(b) Synchronous

Figure 4.15: PI distribution for an iterated AXPY-kernel on System 2. Between each copy cycle, the kernel was launched 100 times. The workload was initialized equally and scheduled by the WP-scheduler with $\omega = 0.5$. (a) For mapped memory, we see no difference compared to one kernel launch. The PI distribution is still the same as in Figure 4.14(e). (b) For a synchronous copy cycle around the iterated kernel, we see a clear shift towards device0. The PI distribution is identical to the static device memory distribution (see Figure 4.13(a)).

Chapter 5

Discussion and future work

We derived multiple dynamic and semi-static schedulers and compared their behaviour for different workloads. Our PI-based scheduling approach requires a few iterations to estimate the optimal distribution. The WP-scheduler is a bit slower at estimating the optimal distribution but using previous run-time data leads to more stable behaviour. Furthermore, the draw-back is less relevant if we make a good initial guess. However, after the optimal distribution is reached the dynamic and semi-static schedulers yield comparable results for an exclusively loaded system.
Then, we turned our attention toward memory management. For the memory-bound kernels, the PI distribution is tied to the bus and component bandwidth. In contrast, for compute-bound kernels, the peak performance is a good indicator for the optimal distribution. Mapped memory has the advantage of not occupying device memory. But if the data is accessed multiple times during kernel execution it is more efficient to manually transfer the data to the device. Modern GPUs allow a noticeable performance increase by the overlapping kernel and copy calls using asynchronous data transfer.

There are several ways to expand our simple model. We only used a single thread for host computation. So the host performance can be improved by parallelizing its kernel. However, we do not expect a significant difference in the PI distribution for compute-bound workloads. Our test system only contained components that were directly connected by the PCIe bus. For a system where components are connected via a network and data transfer is more expensive the question is if our PI estimation is sufficient or if a more complex estimation model which explicitly considers the data transfer time is needed.

Appendix

Bibliography

[ABA12] A. Acosta, V. Blanco, F. Almeida. Towards the Dynamic Load Balancing on Heterogeneous Multi-GPU Systems. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 646–653. IEEE Computer Society, 2012. (Cited on page 11)

[ABA13] A. Acosta, V. Blanco, F. Almeida. Dynamic load balancing on heterogeneous multi-GPU systems. *Computers & Electrical Engineering*, 37:2591–2602, 2013. (Cited on page 11)

[BCV09] J. Bahi, R. Couturier, F. Vernier. Synchronous Load Balancing on Asynchronous Iterative Computation. *Journal of Algorithms & Computational Technology*, 3:135–153, 2009. (Cited on page 11)

[Bin13] A. P. D. Binotto. *A Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi- and Many-Core Desktop Platforms*. Ph.D. thesis, Technische Universität Darmstadt, 2013. (Cited on page 11)

[Boy13] M. Boyer. *Improving Resource Utilization in Heterogeneous CPU-GPU Systems*. Ph.D. thesis, University of Virginia, 2013. (Cited on page 11)

[CK88] T. L. Casavant, J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988. (Cited on pages 13 and 14)

[CLR11] D. Clarke, A. Lastovetsky, R. Reddy. Dynamic Load Balancing of Parallel Computational Iterative Routines on Highly Heterogeneous HPC Platforms. *Parallel Processing Letters*, 21(2), 2011. (Cited on page 34)

[CVKG10] L. Chen, O. Villa, S. Krishnamoorthy, G. R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12. 2010. (Cited on page 41)

[DL15] J.-F. Dollinger, V. Loechner. CPU+GPU Load Balance Guided by Execution Time Prediction. In *Fifth International Workshop on Polyhedral Compilation Techniques (IMPACT 2015)*. 2015. (Cited on page 11)

[DW03] M. Drozdowski, P. Wolniewicz. Out-of-Core Divisible Load Processing. *Parallel and Distributed Systems, IEEE Transactions on*, 14:1048– 1056, 2003. (Cited on page 34)

[GABC08] I. Galindo, F. Almeida, J. Badía-Contelles. Dynamic Load Balancing on Dedicated Heterogeneous Systems. pp. 64–74. 2008. (Cited on page 11)

[GAGZ$^+$20] T. Geng, M. Amaris Gonzalez, S. Zuckerman, A. Goldman, G. Gao, J.-L. Gaudiot. PDAWL: Profile-based Iterative Dynamic Adaptive WorkLoad Balance on Heterogeneous Architectures. 2020. (Cited on page 33)

[GBHS12] C. Gregg, M. Boyer, K. Hazelwood, K. Skadron. Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data, 2012. (Cited on page 11)

[Hag97] T. Hagerup. A Hybrid Dynamic Load Balancing Algorithm for Distributed System. *Journal of Parallel and Distributed Computing*, 47:185–197, 1997. (Cited on page 11)

[HC] W.-M. Hwu, S. Chandrasekaran. GPU-Teaching Kit - Accelerated Computing. URL https://engineering.purdue.edu/~smidkiff/ece563/NVidiaGPUTeachingToolkit/Mod14DataXfer/Mod14DataXfer.pdf. Retrieved July 14, 2020. (Cited on page 52)

[Int] Intel Corporation. Intel Product Specifications. URL https://ark.intel.com/. Retrieved September 30, 2020. (Cited on page 36)

[KM14] M. Katyal, A. Mishra. A Comparative Study of Load Balancing Algorithms in Cloud Computing Environment, 2014. (Cited on page 11)

[Lam15] S. Lammel. CPU-GPU Heterogeneous Computing. 2015. (Cited on page 27)

[LR91] H. Lin, C. Raghavendra. A dynamic load balancing policy with a central job dispatcher (LBC). In *Proceedings 11th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1991. (Cited on page 11)

[LR07] A. Lastovetsky, R. Reddy. Data partitioning with a functional performance model of heterogeneous processors. *The International Journal of High Performance Computing Applications*, 21(1), 2007. (Cited on page 34)

[MIRS14] S. Momcilovic, A. Ilic, N. Roma, L. Sousa. Dynamic Load Balancing for Real-Time Video Encoding on Heterogeneous CPU+GPU Systems. *IEEE Transactions on Multimedia*, 16(1), 2014. (Cited on page 33)

[MJ14] M. Mehta, D. Jinwala. A Hybrid Dynamic Load Balancing Algorithm for Distributed System. *Journal of Computers*, 9:1825–1833, 2014. (Cited on page 11)

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38, 1965. (Cited on page 7)

[Moo75] G. E. Moore. Progress In Digital Integrated Electronics. 1975. (Cited on page 7)

[MR16] M. Mesbahi, A. Rahmani. Load Balancing in Cloud Computing: A State of the Art Survey. *International Journal of Modern Education and Computer Science*, 8:64–78, 2016. (Cited on page 11)

[Nat16] G. Natale. Example of a naive Roofline model, 2016. URL https://commons.wikimedia.org/wiki/File:Example_of_a_naive_Roofline_model.svg. Retrieved September 30, 2020. (Cited on page 23)

[Nvia] Nvidia Corporation. Cuda C++ Programming Guide Version 11.0. URL https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Retrieved July 26, 2020. (Cited on pages 19 and 20)

[Nvib] Nvidia Corporation. Cuda C++ Programming Guide Version 9.1. URL https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_C_Programming_Guide.pdf. Retrieved July 26, 2020. (Cited on pages 9 and 10)

[Nvic] Nvidia Corporation. GeForce GTX 10 Series. URL https://www.nvidia.com/en-us/geforce/10-series/. Retrieved October 5, 2020. (Cited on page 36)

[Pol99] R. Pollak. *Auswirkungen verschiedener Informationsebenen auf die Effizienz der dynamischen Lastbalancierung*. Ph.D. thesis, Universität Stuttgart, 1999. (Cited on page 15)

[Ros19] M. Roser. Moore's Law Transistor Count 1971-2018, 2019. URL https://commons.wikimedia.org/wiki/File:Moore27s_Law_Transistor_Count_1971-2018.png. Retrieved July 30, 2020. (Cited on page 8)

[Sah13] B. Sahoo. Dynamic load balancing strategies in heterogeneous distributed system. 2013. (Cited on page 11)

[SK11] J. Sanders, E. Kandrot. *CUDA BY EXAMPLE- An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011. (Cited on pages 17 and 36)

[SNO$^+$11] M. Shahsavari, M. Nadeem, S. A. Ostadzadeh, Z. Al-Ars, K. Bertels. Task Scheduling Policies in General Distributed Systems: A Survey and Possibilities. 2011. (Cited on page 13)

[SP94] M. Srinivas, L. M. Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, 1994. (Cited on page 15)

[VD08] V. Volkov, J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. 2008. (Cited on page 24)

[XL95] C.-Z. Xu, F. Lau. Iterative Dynamic Load Balancing in Multicomputers. *The Journal of the Operational Research Society*, 45, 1995. (Cited on page 11)

[ZXZ$^+$17] C. Zhang, Y. Xu, J. Zhou, Z. Xu, L. Lu, J. Lu. Dynamic load balancing on multi-GPUs system for big data processing. In *2017 23rd International Conference on Automation and Computing (ICAC)*, pp. 1–6. 2017. (Cited on page 11)

Declaration