

AMERICAN UNIVERSITY OF BEIRUT

ENERGY AWARE SCHEDULER FOR CLOUD COMPUTING  
TASKS IN A DATACENTER ENVIRONMENT

by  
NIZAR RABIH EL ZARIF

A thesis  
submitted in partial fulfillment of the requirements  
for the degree of Master of Engineering  
to the Department of Electrical and Computer Engineering  
of the Faculty of Engineering and Architecture  
at the American University of Beirut

Beirut, Lebanon  
December 2014

AMERICAN UNIVERSITY OF BEIRUT

ENERGY AWARE SCHEDULER FOR CLOUD COMPUTING  
TASKS IN A DATACENTER ENVIRONMENT


by  
NIZAR RABIH EL ZARIF

Approved by:

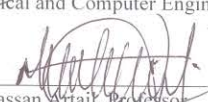


\_\_\_\_\_  
Dr. Mariette Awad, Assistant Professor  
Electrical and Computer Engineering

\_\_\_\_\_  
Advisor

  
\_\_\_\_\_  
Dr. Ayman Kayssi, Professor  
Electrical and Computer Engineering

\_\_\_\_\_  
Member of Committee

  
\_\_\_\_\_  
Dr. Hassan Artail, Professor  
Electrical and Computer Engineering

\_\_\_\_\_  
Member of Committee

Date of thesis defense: December 19, 2014



# AN ABSTRACT OF THE THESIS OF

Nizar Rabih El Zarif for Master of Engineering  
Major: Machine intelligence

Title: Energy Aware Scheduler For Cloud Computing Tasks In A Datacenter Environment

The average power consumption per datacenter is around 1 MW per year, making the power consumption of one datacenter equivalent to that of a small town. It is estimated that the state-of-the-art datacenter consumes around 0.8W to cool down 1W of heat generated by a server. Also, the cooling cost makes up 15% of the total cost of ownership. A survey by J. Koomey indicated that nearly 5000 MW were consumed by datacenters in the US alone in 2005, costing around 2.7 billion dollars in electric bills. Hence, the need for a better power management arises. Most of today's datacenters use either Least Loaded First or Round Robin scheduling algorithms which result in and large energy consumption.

To reduce the energy cost of operating a datacenter, we look into efficiently scheduling the workload among the available servers. Thus, we modeled the workload scheduling problem as a Variable Cost and Size Bin Packing Problem, and introduced two new solutions based on the Best Fit algorithm. The first solution - Divide and Conquer Best Fit- is a modified version of the Best Fit algorithm optimized for multicore processors. The second solution is the Accelerated Best Fit optimized specifically for a Graphical Processing Unit scheduler. Both algorithms solve VCSBPP and reduce the energy consumed in datacenter servers 7000 times faster than BF. This translates in transforming BF from being a mostly offline solution to an online one.

## CONTENTS

ABSTRACT	VIII
ACKNOWLEDGMENTS	XII
LIST OF ILLUSTRATIONS	XIII
LIST OF TABLES	XIV
LIST OF ABBREVIATIONS	XV
Chapter	
1. INTRODUCTION	1
2. RELATED WORK	5
2.1 Datacenter optimization techniques.....	5
2.1.1 Architectural Techniques.....	5
2.1.2 Virtualization Techniques.....	6
2.1.3 Energy and Temperature Aware Scheduling Techniques.....	7
2.2 Bin Packing Problem.....	7
2.3 Datacenters and the Cloud.....	9
2.3.1 Cloud computing.....	10

2.3.2 Cloud Services .....	11
2.3.3 Power Model of a Server .....	12
2.4 Compute Unified Device Architecture .....	13
2.4.1 GPU Architecture.....	14
2.4.2 CUDA Programming Model.....	17
2.4.3 CUDA Memory Model.....	19
2.4.4 Additional CUDA Programming concepts .....	21
<b>3. PROPOSED TASKS ALLOCATION IN</b>	<b>23</b>
3.1 Service Level Agreement Violation .....	24
3.2 Mathematical Model.....	24
3.3 Proposed Cost Function.....	25
3.4 Bin Packing Problem .....	26
3.5 Best Fit As a Scheduler.....	28
3.6 Divide and Conquer Best Fit .....	33
3.7 Accelerated Best Fit.....	37
3.8 Scalability .....	42
<b>4. SETUP AND TESTING</b>	<b>44</b>
4.1 Results and Discussion .....	47

5. CONCLUSION	58
REFERENCES	59

## ACKNOWLEDGMENTS

First I would like to praise god for his constant blessings and gifts.

Second, I would like to thank my professor and advisor Dr. Mariette Awad for her constant help and supervision. Also I would like to thank Prof. Mohammed Adnan Alaoui for his help in CUDA and OpenMp.

Third, I would like to thank my friends and family for their constant technical, moral and writing support throughout this thesis.

This work is supported by MER, a partnership between Intel Corporation and King Abdul-Aziz city for science and technology (KACST) to conduct and promote research in the Middle East and by the university research board at the American University of Beirut.



# ILLUSTRATIONS

## Figures

1: Servers with CRAC Units [4].....	2
2: cloud architecture [24].....	11
3: CPU and GPU architecture [36].....	14
4: GPU architecture [37].....	15
5: Anatomy of an SM [37].....	16
6: CUDA Programming Hierarchy WHERE IS THE REFERENCE? .....	17
7: Centralized Scheduler.....	29
8: Virtual Local Schedulers .....	34
9: Total Energy in Data Centers.....	47
10: Logscale execution time .....	49
12: System 2 logscale execution time.....	53

## TABLES

### Tables

1: The characteristics of the four types of servers considered.....	46
2: Energy reduction percentile compared to round robin .....	48
3: Times Speedups compared to Best Fit.....	50
4: SLAV for fixed T and variable N and M.....	51
5: Energy Reduction Percentile .....	52
6: SLAV .....	53
7: The relative performance increase compared to BF .....	54
8: SLAV vs arrival rate for N=20000, M=2000 .....	55
9: SLAV vs batch period for N=40000, M =4000 .....	56
10: SLAV vs batch period for N=100000, M=10000.....	56
11: SLAV vs batch period for N=200000, M=20000.....	57

## ABBREVIATIONS

Abbreviations:

**ABF:** Accelerated Best Fit

**API:** Application Programmable Interface

**BF:** Best Fit

**BFD:** Best Fit Decreasing

**BPP:** Bin Packing Problem

**DCBF:** Divide and Conquer Best Fit

**DRAM:** Dynamic Random Access Memory

**CPU:** Central Processing Unit

**CUDA:** Compute Unified Device Architecture

**DVFS:** Dynamic Voltage Frequency Scaling

**FCFS:** First Come First Served

**FF:** First Fit

**FFD:** First Fit Decreasing

**GFLOPS:** Giga Floating Point Operation per Second

**GPU:** Graphical Processing Unit

**GPGPU:** General Purpose Graphical Processing Unit

**LLF:** Least Loaded First

**SLAV:** Service Level Agreement Violation

**SM:** Streaming Multiprocessor

**SP:** Single Processor

**SRAM:** Synchronous Random Access Memory

**RAM:** Random Access Memory

**RR:** Round Robin

**GDDR:** Graphical Double Data Rate

**PSU:** Power Supply Unit

**PCIe:** Peripheral Component Interconnect Express

**VCSBPP:** Variable Cost and Size Bin Packing Problem

**VM:** Virtual Machine

# CHAPTER 1

## INTRODUCTION

The average power consumption per datacenter is around 1 MW per year, making the power consumption of one datacenter equivalent to that of a small town [1]. It is estimated that the state-of-the-art datacenter consumes around 0.8W to cool down 1W of heat generated by a server. Also, the cooling cost makes up 15% of the total cost of ownership [2]. A survey by J. Koomey indicated that nearly 5000 MW were consumed by datacenters in the US alone in 2005, costing around 2.7 billion dollars in electric bills [3]. Hence, the need for a better power management arises.

Most of today's datacenters use either Least Loaded First or Round Robin scheduling algorithms which result in large energy consumption. The online nature of today's services, whether for online banking, online shopping, video steaming, data storage, social networking or messaging requires a continuous stream of power with minimal power losses. These services are provided by a large number of servers in datacenters that are required to be mostly ON around the clock and ready to process any request. Any outage can interrupt ongoing operations and cause a huge impact on business. These servers consume a large amount of power and generate a lot of heat. Reducing power consumption on a server or cluster of servers would reduce generated heat, which in turn will reduce the pressure on the Computer Room Air Conditioning

(CRAC) units. Figure 1 shows airflow and CRAC placement in datacenters [4].

### Hot Aisle/ Cold Aisle Approach

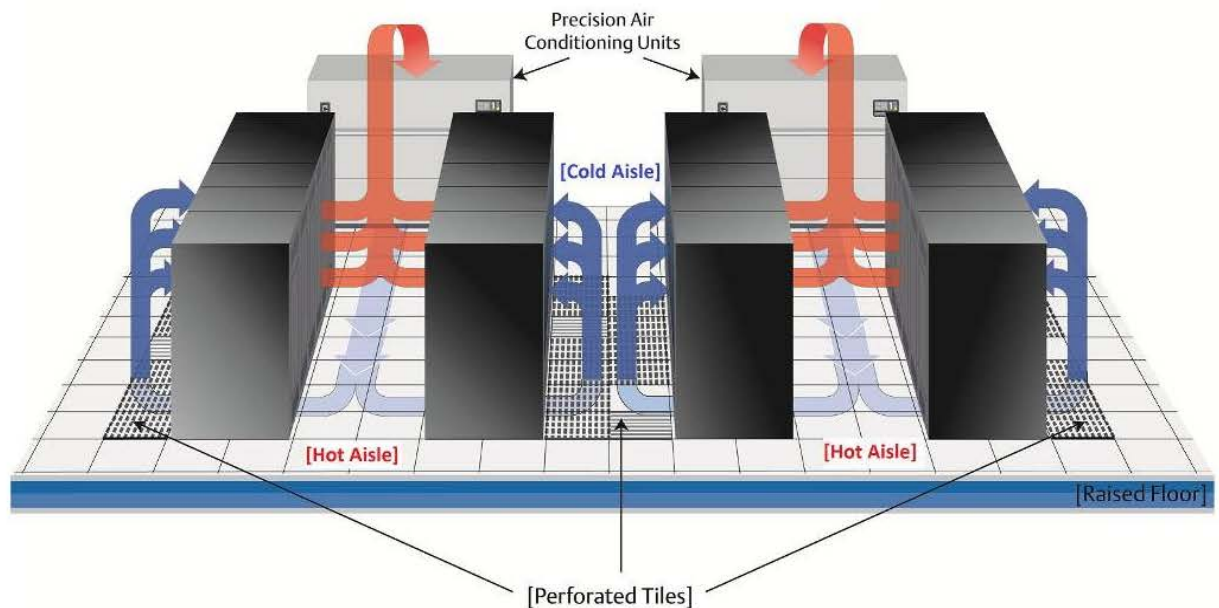


Figure 1: Datacenter Layouts [4]

The demand for servers in a datacenter grew by 56 % between 2005 and 2010 [5] and this trend is likely to continue in the future. Thus, energy-saving techniques are quickly becoming a necessity to reduce the cost of operation and to make datacenters more environmentally friendly. This can be done at different levels: circuits level, algorithms level and virtualization level. Thermal aware workload scheduling is one algorithmic technique that helps reduce the energy consumed by the CRAC units [6]. The row and column position of the server node within a datacenter can affect the temperature of the server within a datacenter, and thus thermal aware scheduling can reduce energy consumed by up to 25% by distributing the workload across multiple servers within a datacenter [6]. Some electrical companies reduce the cost per KWh during off-peak hours and thus further savings can be achieved by rescheduling high demanding tasks to off peak hours [7].

The rest of the thesis is organized as follows:

This research seeks to reduce energy consumption of datacenters by proposing an energy aware task scheduler. Its contribution can be summarized by three main aspects which are:

- 1) Model and design of a simulator that is both accurate and simple,
- 2) Propose a solution for energy aware placement that doesn't sacrifice performance
- 3) Propose two novel and very fast solutions for Variable Cost and Size Bin Packing Problem (VCSBPP): one optimized for multicore CPU and one for GPU

The rest of this thesis is organized as follows:

## **Chapter Two: Related Works**

A survey of the available architectural, virtualization, thermal and power-aware saving techniques for datacenters is presented in this chapter along with the Bin Packing Problem (BPP) and its solutions. A basic energy and power model of the datacenter is also presented to help evaluate the power saving techniques proposed in this thesis.

Since we will be using Compute Unified Device Architecture (CUDA) to enhance the performance of our proposed task allocation algorithm, this chapter also includes an overview of CUDA.

## **Chapter Three: Enhancing Task Allocation in Datacenter**

In this chapter, the task allocation problem is presented and modeled as a VCSBPP problem. The most known algorithm for solving the VCSBPP problem – Best Fit (BF) – is discussed and its limitations are highlighted. And then, two proposed improvements on BF – Divide and Conquer Best Fit (DCBF) and Accelerated Best Fit (ABF) – are presented.

## **Chapter Four: Setup and Testing**

This chapter explains the setup used to verify the proposed task allocation techniques. Also, simulation results are also provided and analyzed.

## **Chapter Five: Conclusion**

Finally, this chapter concludes the thesis with a summary and possible future works.



## CHAPTER 2

### RELATED WORK

Different techniques were introduced in the literature to optimize datacenters with respect to performance, energy or temperature. This section will start by presenting the work done on reducing energy through circuits and architectural techniques. Next, the virtualization techniques to reduce the number of servers required in a datacenter and thus the energy consumed are presented. We also introduce the server power, and finally an introduction to CUDA that will help clarify few concept used in ABF.

#### **2.1 Datacenter optimization techniques**

##### *2.1.1 Architectural Techniques*

On the architecture level, several techniques improve the performance given a constrained power budget [8], or reduce energy consumption without significantly degrading the performance [9].

[8] proposed the MaxJobPerf algorithm which finds the least demanding tasks and schedules them first. Next, the algorithm determines the optimal frequency and voltage to use for each task. This step assumes the server has a DVFS controller that is capable of varying the frequency and voltage. MaxJobPerf guarantees the server is kept busy without exceeding the power budget and shows up to 50 % lower wait time to execute benchmarks. Since the energy consumed has a quadratic dependence on voltage, DVFS significantly reduces the energy consumed to finish one operation. However, reducing the voltage necessitates a reduction in frequency which could degrade performance if not used correctly.

In [9], the authors suggest reducing power of the CPU by reducing the power consumed by mispredicted instructions. The CPU pipeline must be flushed when the branch predictor fails to correctly predict the result of a branch. Flushing the pipeline contributes to about 28% of the total power consumed by the system. Because throttling of the fetch and decode reduces the bandwidth of their respective units and might cause stalling of the units altogether, the authors applied selective throttling in the fetch and decode stage based on the confidence level of the branch predictor. For low confidence branches, the authors suggest using aggressive throttling; whereas, less aggressive throttling is used for high confidence branches. Their method of selective throttling resulted in 18% energy reduction.

### *2.1.2 Virtualization Techniques*

The authors in [10] proposed three techniques of VMs live migration to reduce energy consumption in datacenters: minimizing migration, highest potential growth and random search techniques. Minimizing migration technique allows VM migration when the upper and lower CPU utilization are met. Highest potential growth technique allows migration when the CPU utilization cost is least. Random search technique will only allow migration of VM based on uniform probability distribution. The VMs are then assigned to other servers using a modified version of BFD algorithm which determines the cost of placement of VM in every possible location. Their simulation showed a maximum of 87% power reduction compared to the non-power aware policy where no power saving technique, hardware or otherwise was used.

### *2.1.3 ENERGY AND TEMPERATURE AWARE SCHEDULING TECHNIQUES*

There have been many approaches in literature for task scheduling and resource allocation. Energy aware task placement such as the ones proposed in [11] and [12] aim at reducing the total server energy required to run the tasks. Temperature aware scheduling such as in [2] and [6] aim at reducing the overall cooling costs. Performance aware scheduling such as in [13] aim at getting the most performance out of the datacenter, or reducing the serving time as in [14], or improving the performance per single machine such as in [15].

Many of these approaches tend to neglect the execution time of their algorithm. Since most of these algorithms are computationally expensive, most applications tend to favor a static scheduling approach, where the algorithms are run offline. However, in datacenters dynamic scheduling is required. A dynamic scheduling technique is capable of getting the best allocation possible under a tight time constraint, preferably much lower than the arrival rate of tasks.

The work presented in this thesis is closest to the work done in [10] which the authors modeled the task scheduling problem in datacenters as a BPP and solved the task scheduling using BFD. While their approach yielded significant energy reduction, it tends to take a large amount of time to find the optimal placement which might make it infeasible for online use.

## **2.2 Bin Packing Problem**

The traditional BPP is defined as packing a list of items  $L = \{l_1, l_2 \dots, l_L\}$ , with volume  $V = \{v_1, v_2 \dots v_L\}$ , into the minimum number of uniformly sized bins  $B = \{b_1, b_2 \dots b_3\}$ . A typical solution to the BPP tends to cluster items in the least amount of bin possible to reduce the number of required bin. This is somewhat similar to the desired energy reduction algorithm in datacenters since reducing the amount of active

servers needed to run the tasks without overloading any server will reduce the overall energy consumed by the datacenter without significantly impacting its performance.

There are three widely used methods to solve the traditional BPP: First Fit Decreasing (FFD), BF and Best Fit Decreasing (BFD) algorithms. In FFD algorithm, the items are sorted according to their volume and then each item is placed into the first bin that could hold it. This method is fast and efficient but does not give the best results, especially in the case of non-homogeneous size bins. The BF algorithm tests the cost of placing the item in all possible bins, and then assigns it to the bin that has the least cost. The BFD algorithm is similar to the BF algorithm but starts with sorting the items according to their volume. BF and BFD work well for both homogeneous and non-homogeneous bins, but they tend to be computationally expensive

Other solutions for BPP and its variation were also proposed. The authors in [16] used a randomized pre assignment technique and improved the solution using branch bound algorithm. The authors of [17] proposed a solution to VCSBPP by fixing the number of bins and transforming the problem into a knapsack problem. Authors of [18] introduced Geometric Heuristic which is a variation of BFD by using a cost function that measures the correlation between bins and the remaining tasks in the queue. The authors of [19] used a genetic algorithm-based approach for allocating tasks to solve a three dimensional BPP. This approach tends to give a good local solution in a short amount of time. However, the local solution might not be the optimum solution and this algorithm might give a worse result when an optimal solution exists and is feasible to find. The authors of [20] introduced two algorithm variations of the BF algorithm - Harmonic Match and Refined Harmonic Match - that divide the search space into opposing quadrants and match these quadrants into bins using BF. Their method tends to give better results than BF alone in the worst case.

From the literature we can see that BF and its variant are the most used solution for the coding simplicity and their capability to give good results in most

cases. Their big drawback is their large computation time that makes them inviable for online use when the problem size is large.

## 2.3 Datacenters and the Cloud

Most scheduling techniques presented in the literature deal with only one parameter, either power or performance or temperature. However, in today's datacenters there is a need to optimize all three parameters (power, performance and temperature) to ensure that datacenters stay within the power budget while providing a good quality of service. A dynamic scheduler must have an execution time much lower than the arrival rate of tasks, and at the same time provide an optimal or near optimal placement. The time of execution is a key element that most of the literature tends to leave out. Since most of the previously proposed algorithms are too computationally expensive, the time required to run these algorithms is too high to be used online in datacenters if the arrival rate of tasks is large. As a result, datacenters avoid using these computationally expensive schemes since the time required to get an optimal allocation is not practical for online use.

From the literature, we know that workload prediction is possible and it allows the scheduler to estimate the best possible placement for application performance or for reduced energy consumption. A smart allocation minimizes power consumption while maximizing performance can lead to a good quality of service while also reducing cost on the service providers.

The performance counters that can be used for performance and temperature prediction are: instruction mix, branch prediction accuracy, data cache hit and miss, instructions per cycle and periodicity of phases [21]. However, the computational complexity of some of these counters prohibits their use. Some of these metrics, however, can be retrieved using performance monitoring application programming

counter. It has been shown in [21] that the best time difference between consecutive predictions is 10ms. Prediction is achievable since program phases are periodic. There are two types of predictors: Simple statistical predictors such as: Average (N), last value, Exponentially Weighted Moving Average, and History predictors. History predictors include hash tables with fixed size predictors that are better for variable workload and run length encoded history that are more suitable for stable behavior. However, one must note that using a limited hash table would decrease prediction accuracy and an accurate predictor can be used to improve system performance with minimal impact on power [22]. Once the performance ranking is performed for all nodes, ranking job migration starts. Any upcoming task should be scheduled with the best ranking. Also, a reasonable approach is to have several strategies depending on the demand such as performance oriented policy, power conservation policy and general policy.

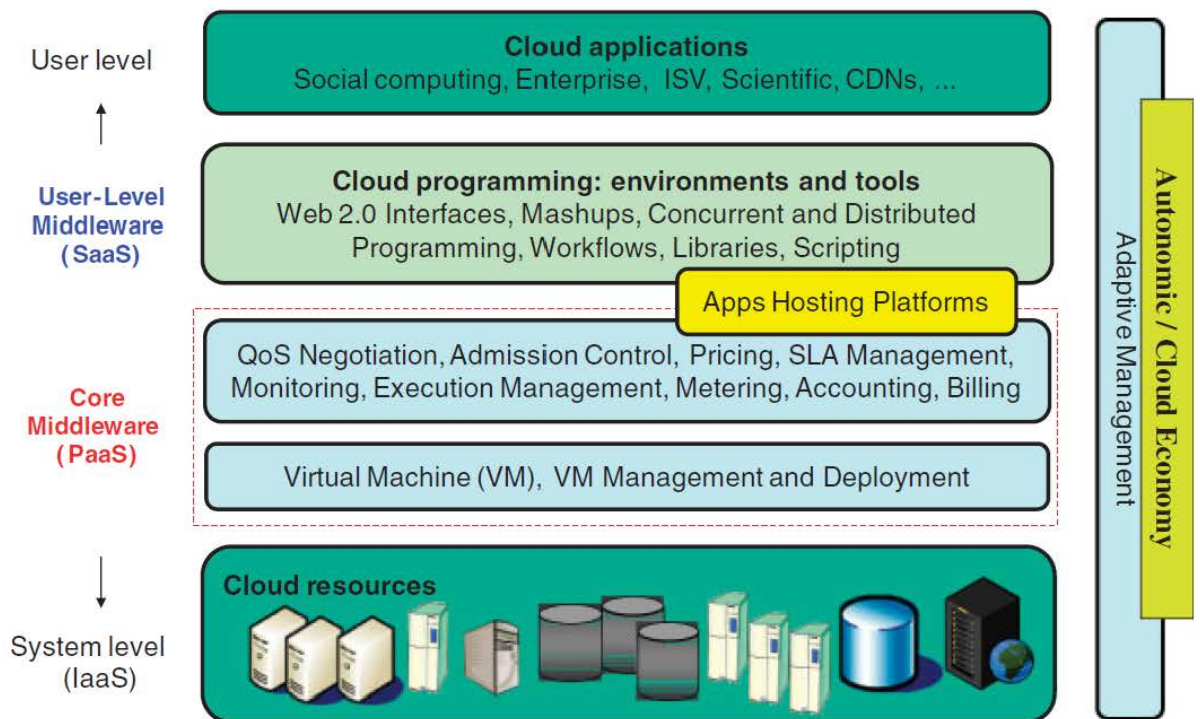
Temperature and placement aware systems can help with getting more distributed temperature. Undistributed temperature might lead to lower performance – due to temperature capping – and lower energy efficiency – due to development of heat points inside the datacenter. A system where neighbors have high temperatures would cause the system temperature to rise because of convection rather than real workloads. It is wiser not to select such a system to perform heavy tasks since it might cause rapid temperature build up in that area of the datacenter. Since the CPU accounts for most of the server power [23], optimizing the scheduling and migration policy is performed while taking CPU power into account.

### ***2.3.1 Cloud computing***

A common name for internet or datacenter services is cloud computing, as the user is unaware about the number of computer or the networking behind the services. The cloud architecture consists of four layers as shown in Figure 2. The cloud user

application resides in the highest layer providing the service for the consumer. The user-level middleware resides in the second layer providing the interface for the user level applications through the user library. Below the user level application and services resides the core level middleware providing the necessary messaging across the platforms and monitoring the different resources. This layer also provides the VM services and VM management. The last level in the system provides the physical requirement for the cloud such as computation and storage [24].

#### CLOUDSIM: A TOOLKIT



**Figure 2: cloud architecture [24]**

#### 2.3.2 Cloud Services

Cloud computing is becoming more relevant by the day with new platforms and new services being launched on a regular basis. The application ranges from IT professionals to basic users, with all backend processing happening in large datacenters invisible to users. Some of the more popular services are: remote IT services such as Amazon Elastic Compute Cloud [25], online storage services such as iCloud [26],

OneDrive [27], Google Drive [28], DropBox [29] and workload offloading services such as Microsoft Azure [30]. To an online data center operating system management tool such as OpenStack [31].

### 2.3.3 Power Model of a Server

The rapid spreading of cloud applications increases the pressure on the economy of a datacenter due to energy cost, and on the environment due to high CO<sub>2</sub> emissions. The authors of [32] shows that an idle server for a Dell datacenter consumes around 383.75W when idle and 454.39W under stress, due to the high constant power cost and cooling cost. Typically the energy consumption per server in a datacenter is divided into two components: a constant energy and a dynamic energy. The constant energy is consumed by the memory, disks, fans, motherboard and power supply, while the dynamic is consumed by the CPU. Cisco showed that for a typical datacenter under load the power consumption varies around 360W to 380W [33]. The author of [34] did a breakdown of the idle power consumed in a typical datacenter. For an idle server a huge portion of the power goes to the constant power for static component, which account for about 66% of total idle server power while the rest is consumed by the power supply and the CPU. [35] showed that a VM based datacenter can save around 18% to 30% of power just by optimal placement of VMs.

Based on the literature [32-34], the power ( $sp_n$ ) and energy ( $se_n$ ) consumed by a server  $n$  in a datacenter containing  $N$  servers with a total of  $M$  tasks that need to be scheduled can be modeled as follows:

$$su_n \begin{cases} > 0, & sp_n = \alpha_n + \beta_n * \sum up_m^n \\ = 0, & sp_n = sleep_n \end{cases} \quad (1)$$

$$if \ su_n \begin{cases} > 0, & se_n = \sum_{m=0}^M (\alpha_n * up_m^n + \beta_n * up_m^n * ut_m^n) \\ = 0, & se_n = sleep_n * 1/\mu \end{cases} \quad (2)$$



where  $su_n$  represents the utilization of server  $n$  which cannot be less than 0,  $se_n$  represents the energy of server  $n$ ,  $up_m^n$  represents the task  $m$  processing utilization that has been assigned to server  $n$ ,  $ut_m^n$  represents the task  $m$  processing time requirement that has been assigned to server  $n$ ,  $\alpha_n$  is the power constant consumed by the server  $n$ ,  $\beta_n$  is the efficiency coefficient (a small  $\beta_n$  represent an efficient system),  $u_n$  represents the utilization of the system,  $\mu$  is the arrival rate of tasks and  $sleep_n$  is the power consumed by the server  $n$  when it is in sleep mode.

For energy reduction a minimal number of servers must be used. Allocating too many tasks per server can result in reduced energy consumption at the expense of application slowdown or failure. This is an undesirable side effect that should be taken into account when running any energy saving algorithm. From the equations, one can deduce that the power and the energy consumed by the server is highly depend on what has been assigned to it and the arrival rate of the tasks.

## 2.4 Compute Unified Device Architecture

CUDA is a parallel computing programming platform that allows accelerated processing by offloading some of the work from the CPU to the GPU. CUDA is available on many modern Nvidia GPU, to enable users to take advantage of the massively parallel capability of the GPU for a non-graphical workload. The CUDA programming model was built on top of well-developed languages such as C/C++ and FORTRAN with special API that abstract some of the complexity from the developer.

GPUs are designed to provide high throughput by implementing many cores that work at lower frequencies compared to the CPU, making them ideal for an “embarrassingly parallel” application – such as matrix multiplication. However, due to the parallel nature of GPUs, implementing highly serialized algorithms such as BF would be extremely inefficient and inherently slower than using a CPU.

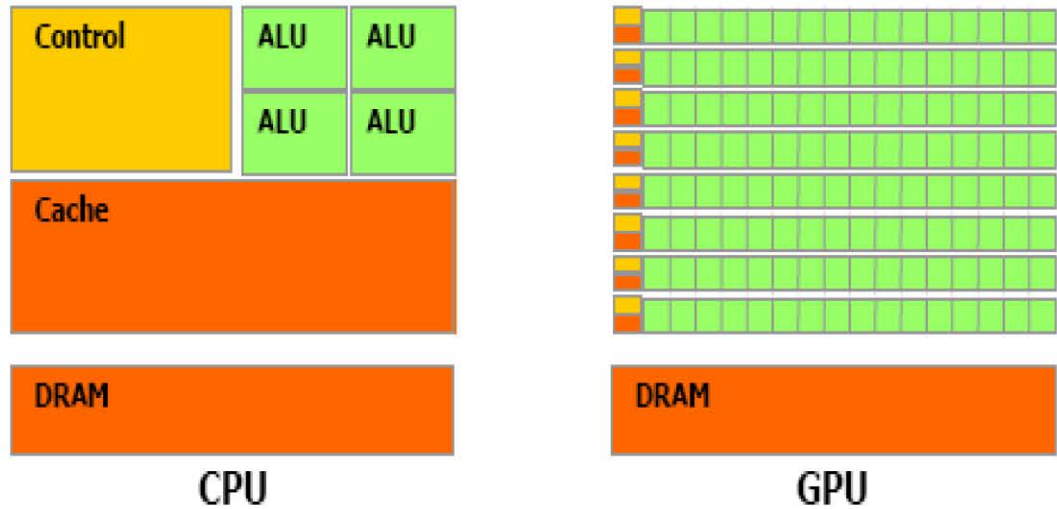


Figure 3: CPU and GPU architecture [36]

Figure 3 shows the high level difference between the CPU and GPU [36].

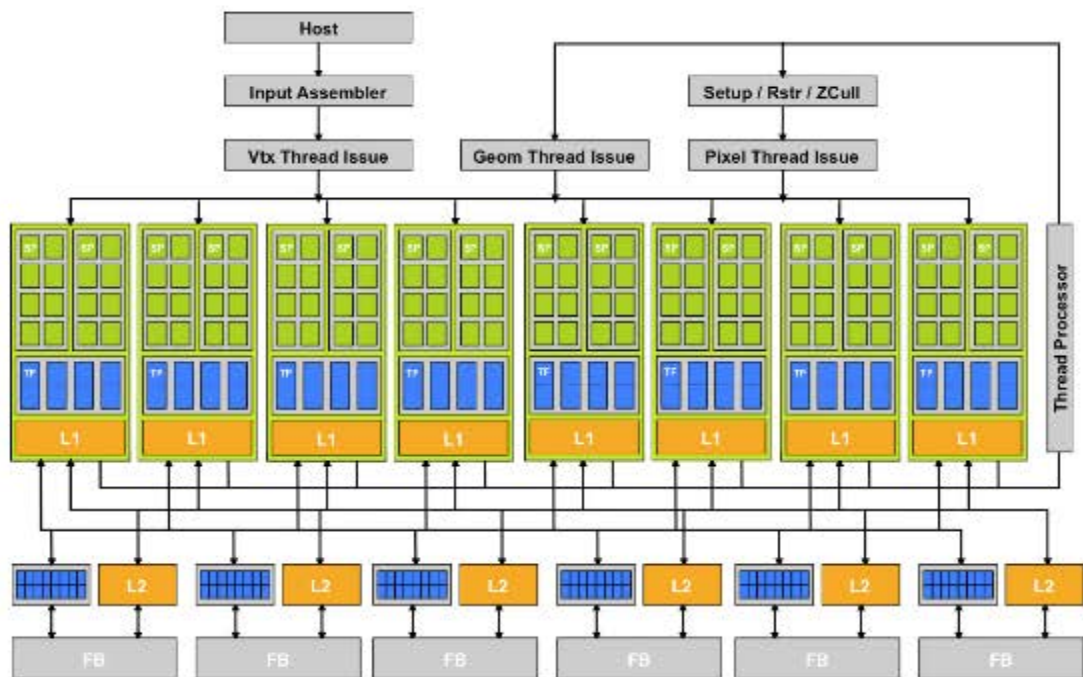
CPUs have larger cache and more complex but fewer Arithmetic Logic Units (ALU) and more complex control.

#### 2.4.1 GPU Architecture

Unlike the CPU design which encompasses a small number of cores with large amount of cache and favors reduced execution time over throughput, the GPU is designed with throughput in mind. Thus, the GPU is built up from a large number of small cores that have a small instruction set. It uses a small shared cache for every group of cores. The small amount of cache is essential for parallel applications to provide speedups. While the GPU is capable of providing significant performance enhancement, it has a number of limitations that programmers must be aware of and that might result in significant degradation in performance if not accounted for. The first limitation is limited shared memory available for every group of cores. Another limitation is the dramatic change in the underlying GPU architecture with every generation of GPUs. For example the Tesla GPU generation is only capable of allocating 16 KB of shared memory per block while the newest Maxwell GPU has

48/64KB of available shared memory. Note that shared memory is one of the four types of memory available for use within a GPU; these types of memory will be explained in the following subsections. A third limitation is the generation dependent APIs available to the programmer. This means that some new APIs might not be available for old GPU. However, newer GPUs can run all codes designed with older API generations more efficiently.

On a high level, the GPU consists of an interface to connect it to the main CPU in the system, a number of streaming multiprocessors (SM) that are connected

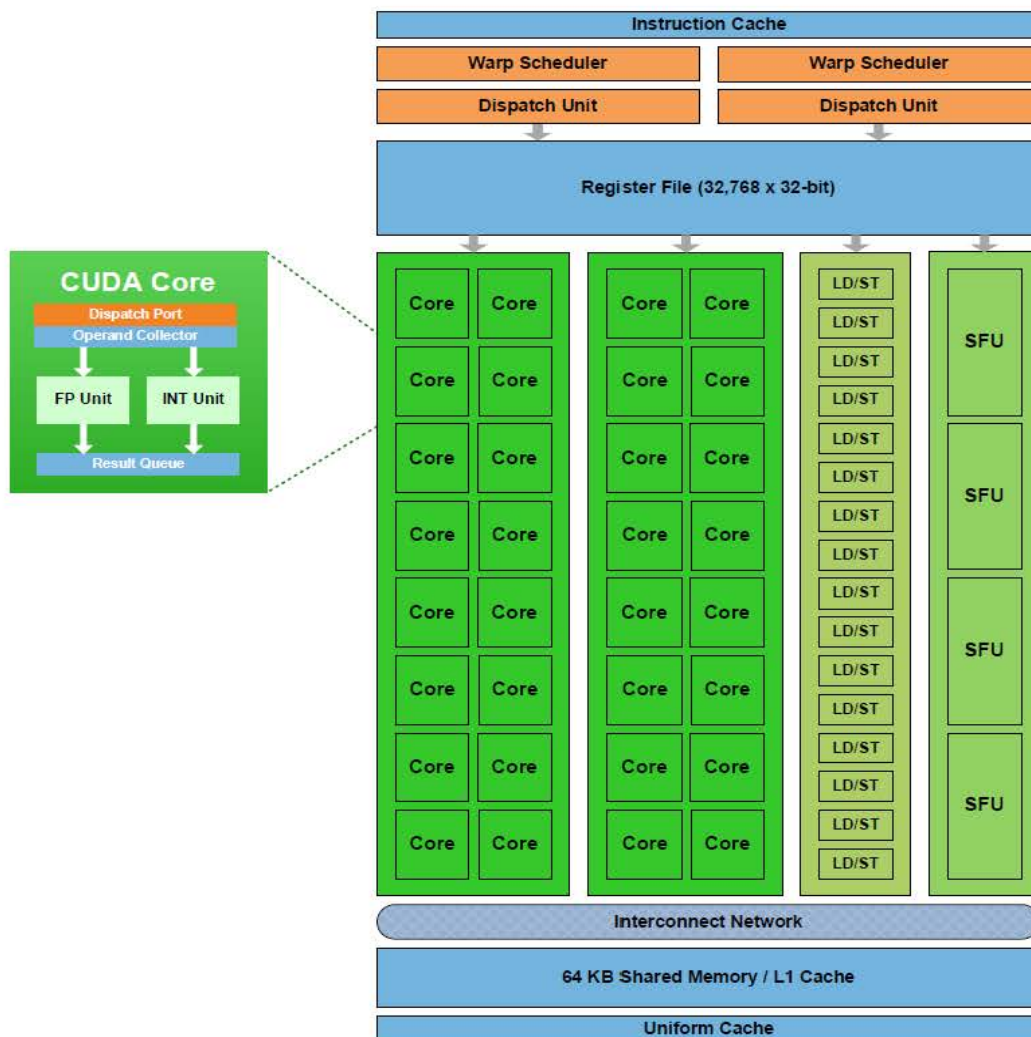


**Figure 4: GPU architecture [37]**

together with a main interconnect network and share a global memory. Figure 4 shows the components making up the GPU. The most notable component of the GPU is the SM [37].

Figure 5 shows the anatomy of the SM block within a GPU [37]. The number of SMs in a GPU ranges from two for low end GPUs to dozens for high end GPUs. Each SM in turn is formed from several Single Processor (SP), a Multithreaded

Instruction Unit (MT Issue, I-Cache, C-Cache in Figure 5), special function units (SFU in Figure 5), double precision unit (DP in Figure 5) and shared memory. Even though the specifics of each of those units varies, a typical SP usually contains integer and single precision floating point execution units and a small register file. The SFU offer more complicated mathematical operations such as trigonometric functions... The DP unit contains a double precision floating point execution unit along with multiply add units.



**Figure 5: Anatomy of an SM [37]**

Each CUDA block can be assigned to one or several MP depending on the resources required for the operation. However a CUDA block cannot be assigned to

more than one SM. The number of CUDA SP, cache size and special functions and number of registers vary by generation.

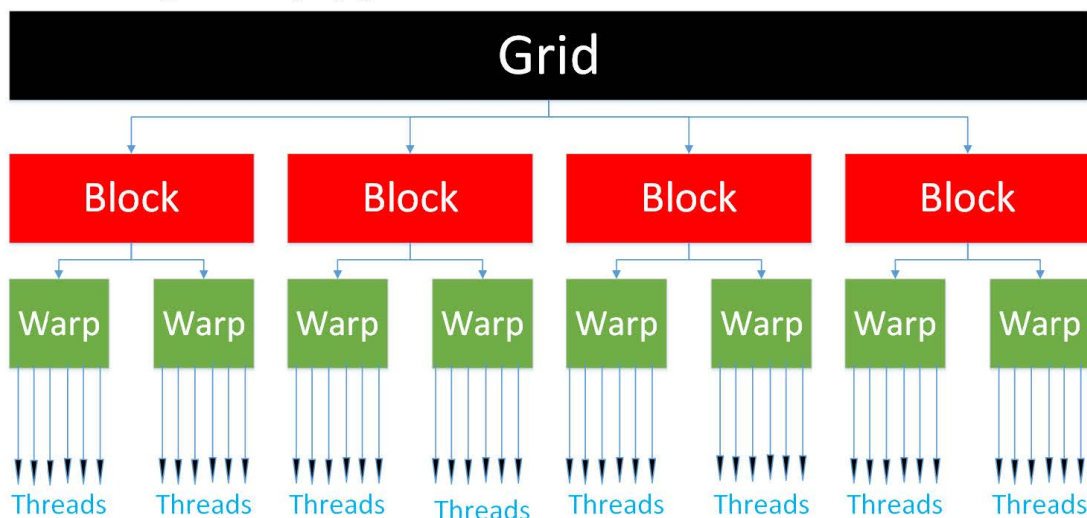


Figure 6: CUDA Programming Hierarchy WHERE IS THE REFERENCE?

#### 2.4.2 CUDA Programming Model

CUDA programming model falls into the follows the hierarchical paradigm presented in figure 6. On the lowest level in the threads, the threads in CUDA are lightweight threads that are abstracted by CUDA API. Threads are created, destroyed or scheduled as needed by CUDA and the programmer cannot control which threads goes to which MP or SM or even CUDA core. CUDA threads are considered “lightweight” because they take a very short time to be created, scheduled and destroyed unlike CPU threads. Each thread has a built in ID that allows it to be called by the programmer, several registers and local memory that are only available to it.

On the second CUDA level warps are found, where warps are group of threads typically 32 consecutive threads. Warps are the execution unit of CUDA, which means that a warp generally runs all the instructions until it finish all operations or reaches a barrier that requires the data to be synchronized with other warps is reached. For best performance, the programmer must insure that as many warps are running at



the same time as possible and avoid thread divergence. A thread divergence incurs when a number of threads inside a warp are executing some instructions while the others threads are executing other instruction due to ( if-else) statement and (switch-case) statement such as:

```
If (Thread_ID < 16)
    Foo();
else
    Bar();
```

In the previous example, when a warp reaches the “if statement” the threads diverge into two group. The first group with ID < 16 takes the “if branch” and others threads are deactivated until the first group to finish executing the branch. After completing the branch, the second group then takes the “else branch” and the first group of threads are deactivated until all other branches are taken. The deactivating and waiting causes slowdowns, which is why thread divergence must be avoided if possible.

The third CUDA level is the Block level. Each block is assigned to one MP inside an SM. If the available resources in an MP are not enough a CUDA block can be assigned to more than one MP, the resources limitation are either threads, number of registers or shared memory size. Different GPU generations have different number of MP and SM and different size of shared memory and different number of registers. Tuning the performance for all generation is difficult but possible. The maximum number of threads in a CUDA block is 512 for “Tesla” GPU generation, and 1024 threads for newer generations, such as “Fermi”, ”Kepler” and “Maxwell”, at the time of writing Maxwell is the newest generation currently available in the market with plans for “Volta” in the future. While the number of threads per block is limited the number of blocks is very high  $64355*64355*64355$  in making it virtually an unlimited total number of threads.

The highest level of CUDA is the grid level. The Grid is formed from a number of CUDA blocks which can be 1D or 2D or 3D arrays. Each kernel defined by the programmer is executed on a grid of threads.

### *2.4.3 CUDA Memory Model*

CUDA provide 4 types of public memory that corresponds to the DRAM memory on the GPU:

- 1- Global Memory (read/write across grids) is available for read and write from all CUDA threads, thus allowing memory sharing for threads, warps and blocks. The global memory is accessible by both the GPU SMs and the CPU.
- 2- Constant memory is a special type of memory that is only available for read from all threads. The size and contents of constant memory is declared and provided by the CPU
- 3- Texture memory is also available for read from all threads with the added advantage of built in local interpreter, which allows automatic estimation. The size and contents of the Texture memory is declared and provided by the CPU.
- 4- Local memory is another type of memory available that is declared by the GPU. It is only available to thread that declared it when the thread variables cannot fit into its local register.

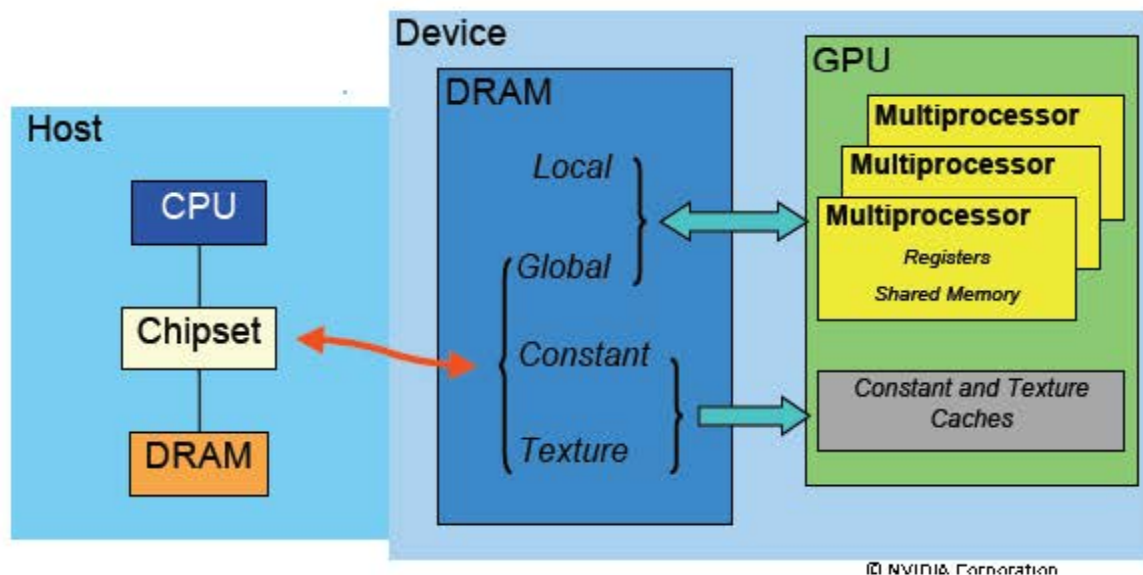
The other type of memory lies in a local level such as:

- Registers are the fastest type of memory available. They exist within the SP block and thus are only accessible to the thread that defined them.

- Shared memory is an SRAM memory that is available per-block, which means every thread in a block can read and write from/to it. Because it is shared it allows some thread cooperation for faster processing, such as Radix add and Radix Min which reduces the computational complexity from  $O(N)$  to  $O(\log_2(N))$ .

Using global memory slows the application speed by a hefty amount since each DRAM access takes about 100 clock cycles while SRAM takes about two cycles

Shared memory, local memory and registers does not have the capability of sending and receiving data from the CPU, hence any memory copy back to the CPU should be done using the Global Memory as shown in figure 7 [38].



**Figure 7: Type of Memory REFERENCE MISSING**

There are two concepts the CUDA programmers must take note off. The first is the thread ID and the second is memory coalescent.



Each thread and block within a grid is assigned an ID and this ID is saved within a register in the SP running the thread. Thus, the ID could be in 1D, 2D or 3D grids.

$$\text{Thread ID} = \text{Local ID} + \text{Block ID} * \text{Block Dimension}$$

To reference the fifth thread in the third block assuming a block size of 128.

The thread ID would be:

$$\text{Thread\_ID} = 4 + 128 * 2 = 260.$$

The Second concept is memory coalescent, which mean any access to the GPU memory should be done sequentially. Coalescent access hides reading and writing latency by copying the data from sequential location in global memory to sequential location in shared memory. A non-Coalescing access might result in poor application performance and can cause crashes. Generally a good approach in handling memory and doing memory intensive operation is by copying the data in coalesced access from the global memory to the local shared memory and any uncoalesced access can be done from the shared memory. The programmer must take into account avoiding back conflict, CUDA API provide some tools such as using atomic operations or thread synchronization for that purpose.

#### ***2.4.4 Additional CUDA Programming concepts***

While there are many more concepts in CUDA that the programmer must be aware of than what is presented above, the section above only mentioned the essential concepts for basic understanding of this thesis. The CUDA setup is relatively easy to acquire since almost any modern Nvidia GPU is capable of performing CUDA operations.

The CPU and GPU code are divided in different sections. The CPU is referred to as host and the GPU is referred as the device. The GPU code is written as a

function within the CPU code and is launched explicitly by the CPU along with the launch parameters. There are three launch parameters which include number of threads per block, the number of blocks, and the size of the dynamic shared memory. The GPU function is referred to as a kernel, so the launch of the GPU function is called kernel launch. The GPU kernel can use special functions called device functions. The host and the device have separate memories, and these memories have to be explicitly allocated, copied and returned if needed. “Kepler” and newer generation GPUs have a unified virtual memory, which means that programmer can allocate a virtual memory that both the CPU and GPU can access, this is done by allocating memory both on the CPU and GPU, and the driver handles automatic updates for any changes in the memory for both the CPU and the GPU memories.

At kernel launch the GPU generates a large number of threads and blocks and allocate the threads and warps to be executed at the available GPU resources. All threads launched will execute the same kernel, the difference between one thread and another is the ID only. The developer should use the ID to make sure that the kernel is choosing the correct data to read and write. This requires a good understanding of the memory model. The terms grid, blocks, threads and ID refer to a logical hierarchy of CUDA, while the term SM and SP refers to the physical hardware.

## CHAPTER 3

### PROPOSED TASK ALLOCATIONS

As mentioned earlier, the standard method of task allocation employed in servers is the *RR* method, which simply assigns a task  $m$  to a server  $n$  where  $n = \text{round}(m/N)$  with  $N$  being the total number of servers in a datacenter. *RR* does not take into consideration the performance and energy impact of this placement and thus results in poor performance and high energy consumption. Nonetheless, it ensures that the maximum number of servers is active at a given time. If the number of tasks ( $M$ ) exceeds  $N$ , *RR* makes sure that the tasks are equally divided among servers. Another commonly employed task allocation method is the *LLF* which allocates the task to the server with the least amount of CPU utilization using *FCFS* policy.

This thesis proposes two new task allocation schemes that optimize both energy and performance within a datacenter. The proposed techniques are also new solutions to *VCSBPP*. These two techniques give significant speed increase compared to *BF* and enable task allocation in a fraction of the time required by *BF*. To facilitate the modeling of the task allocation problem, the following assumptions were made in the proposed techniques. These assumptions are fairly reasonable for modern servers in modern datacenters.

1. The tasks are coming in batches and their execution time and processing requirements are known to the scheduler.
2. The arrival rate of tasks and the number of tasks per batch is also known.
3. The datacenter servers are heterogeneous.
4. The migration time of a VM is assumed to be zero.
5. Any server with zero utilization is assumed to be placed in a sleep state and will consume much less power than when working or idle.
6. All tasks have to be assigned to a server.

7. All tasks have equal priority.
8. All servers are capable of multi-tasking.

### 3.1 SERVICE LEVEL AGREEMENT VIOLATION

The SLAV is a metric that we introduced to measure the overutilization of servers. If the current server utilization ( $su_n$ ) is less than the maximum server utilization ( $sumax_n$ ), the SLAV metric is set to zero. Otherwise, the SLAV per server  $n$  is calculated as the difference between  $su_n$  and  $sumax_n$  multiplied by the time spent in overutilization ( $tmp_n$ ).

$$\text{if } (su_n - sumax_n) > 0, \quad SLAV_n = \sum_0^M (su_n - sumax_n) * tmp_m \quad (3)$$

$$tmp_m = \text{Min}(ut_m, \frac{1}{\mu}) \quad (4)$$

$tmp_m$  is the minimum between the batch time and the task time requirement,  $1/\mu$  is the batch period,  $m$  is the task number and  $M$  is the total number tasks. Ideally SLAV should be equal to zero, which means that the current application will not suffer slowdowns due to less than optimal task placement.

### 3.2 MATHEMATICAL MODEL

Getting the optimal tradeoff between power and performance is a multi-objective optimization problem since we are trying to:

1. **Minimize the total server energy:**

$$\bar{E} = \sum_{n=0}^N se_n \quad (5)$$

2. **Minimize the total SLAV:  $\overline{SLAV}$**

$$\overline{SLAV} = \sum_{n=0}^N SLAV_n \quad (6)$$

**3. Minimize the task allocation execution time ET**

$$ET \ll 1/\mu \quad (7)$$

**4. Subject to  $su_n \leq sumax_n$**

$$su_n = \sum up_m^n \quad (8)$$

### 3.3 PROPOSED COST FUNCTION

To solve this multi-objective optimization problem, we simplify it into a single objective problem, where the number of servers used is to be minimized by consolidating several VMs onto a smaller number of servers. The VCSBPP can then be used to model the VM reallocation problem if the server utilization is assumed to be the item volume and the maximum utilization per server is assumed to be the bin size.

To fit the task allocation problem in datacenters to a BPP, some modification must be made. First, a cost function  $f(n)$  is defined to evaluate the placement of a task  $m$  on server  $n$ . Equation 9 represent the cost function  $f(n)$ . This cost function includes a penalty factor  $K$  which is used to penalize an overutilized server making less likely to be selected as the best server to run the task. This reduced cost function quantifies the impact on both energy and performance of placing task  $m$  on server  $n$ . The task is then assigned to the server with the lowest cost function.

$$f(n) = \begin{cases} up_m * ut_m * \beta_n, & se_n + up_m \leq max_n \\ (\alpha_n - idle_n) * ut_m, & se_n = 0 \text{ and } se_n < max_n \\ (up_m * ut_m * \beta_n * K), & se_n + up_m > max_n \end{cases} \quad (9)$$

### 3.4 BIN PACKING PROBLEM

Assume that we have a batch of tasks  $M$  (list of items) with total processing requirement  $B = \sum_0^m up_m * ut_m$  arriving at every period  $1/\mu$  to a datacenter with  $N$  servers and available processing capability of  $\sum_0^N (sumax_n - su_n)/\mu$ . Initially, all servers are assumed to be unoccupied with tasks ( $su_n = 0$ ) as the scheduling starts,  $su_n$  becomes positive and the available processing is reduced. An intelligent allocation technique should use the available processing power of a datacenter for the best task placement meaning tasks should not over-utilize the servers and at the same time the total energy consumed by the datacenter must be reduced. The solution to BPP provides a good allocation in datacenters since it reduces the number of active servers. In doing so, the non-active servers can be put in a low power mode thus saving energy. However tasks allocation differs from BPP in the ways outlined next.

First servers in datacenters are not homogeneous, some servers are more power efficient while others have more processing capability; some prefer long tasks with small processing requirement while others prefer short and heavy processing tasks. To solve this issue, we are assuming that each task will get a different cost function depending on which server it has been assigned to. Using BPP terminology, we are assuming that each item will take a different volume depending on the bin. This is done by multiplying the volume of the item with an adjustment factor to get the size for that particular bin which is similar to VCSBPP.

Second, a server usually will not be off if its CPU was over-utilized but it will suffer some slowdowns. On the other hand, a physical container cannot hold more than its size. This difference means that even if the server is over-utilized, it could be

assigned more tasks to save power. However, this will cause major slowdowns and thus incur some SLAV. To balance between energy saving and performance degradation, the penalty factor  $K$  within the SLAV definition must be carefully chosen. A small  $K$  sacrifices performance in favor of energy whereas a large  $K$  sacrifices energy in favor of performance.

Finally, BPP prefers reducing the total number of bin used, while this correlate with our allocation scheme, our target is to reduce energy consumption. In some cases it is better to have more low power servers operating than few mid or high power servers, since low power servers might be better for running long operation with small processing requirement. By estimating the energy difference in all possible position we can get a superior result.

The algorithms proposed here are very similar to BF; however some modifications were made to optimize it as a datacenter scheduler:

1. Our cost function corresponds to the total energy and SLAV combined into one value. Our target is to reduce this cost function and not the number of bins as in BF.
2. The numbers of bins here are assumed fixed. Instead of just reducing the number of bin, we want to reduce the number of used bins as long as it correlates with a cost function minimization.
3. To guarantee best performance a high penalty is incurred when the bin is made to hold more than its designed value to guarantee the best performance.
4. The bins are assumed weakly heterogeneous

These proposed modification to BF should give a significant reduction in the number of used bins. However the order of the items in the list is important. Rearranging the item can yield better result but it does come at the cost of higher execution time when the bins are homogeneous which not the case here.

### 3.5 BEST FIT AS A SCHEDULER

BF is used as a scheduler since the task scheduling problem is modeled as a VCSBPP. BF algorithm should give a significant power reduction over *RR* allocation technique. However, this reduction comes at the expense of a higher CPU time. In our simulations the processing time of BF is actually higher than the arrival rates of tasks, which makes this algorithm unfit to be used online. To reduce the CPU time we propose using CUDA and OpenMP to parallelize BF. First, however, we should analyze the algorithm to determine the part that could be parallelized.

Figure 7 shows the BF centralized datacenter scheduler, where the centralized scheduler handles allocating the tasks to all available servers.

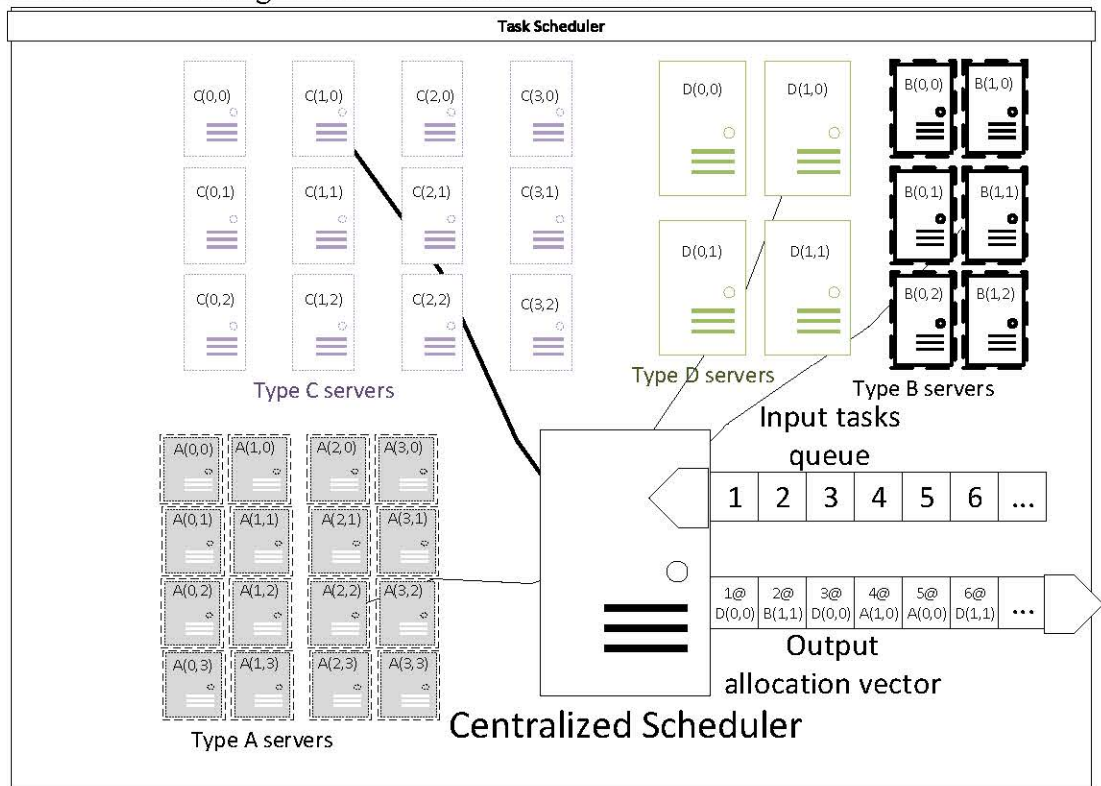


Figure 7: Centralized Scheduler



Assume the following variables:

- $N$  is the total number of servers in a datacenter,  $\subseteq \mathbb{N}$ .
- $t$  is the instance of time,  $t \subseteq [0,1,2, \dots N)$
- $n$  is the server number,  $n \subseteq [0,1,2.. N)$
- $\mu$  is the arrival rate of tasks,  $\mu \subseteq [0, \infty)$ ,  $1/\mu$  is the arrival Period.
- $m$  is the task number,  $m \subseteq [0,1,2 \dots M)$
- $M$  is the total number of tasks,  $M \subseteq \mathbb{N}$
- $up_m^t$  is the task processing requirement of task  $m$  at time  $t$ ,  $up_m^t \subseteq [0,1,2, \dots 100]$
- $su_n^t$  is server  $n$ 's utilization at time  $t$ ,  $su_n^t \subseteq [0,1,2, \dots N)$
- $ut_m^t$  is the required processing time for task  $m$  at time  $t$ ,  $ut_m^t \subseteq [0, 1, 2 \dots 100]$ .
- $a_m^t$  is the allocation vector that describes the location where the task is going to be executed
- $se_n^t$  is the energy consumed in server  $n$  at time  $t$
- $E^t$  is the total energy at time  $t$
- $I$  is the number of iteration where  $I \subseteq \mathbb{N}$
- $i$  is the iteration number and  $i \subseteq [0,1,2 \dots I]$
- $O$  is the number old unfinished tasks, initially  $O = 0$
- $C$  is the combined number of tasks

BF allocation algorithm consists of the following steps:

1. Initialize values  $(I, O, C, M, N, \mu, K)$
2. **Loop1:** for number of iteration  $I$ 
  - a. Generate  $M$  number of tasks based on random number generators

- b. Combine M tasks with previously unfinished O tasks and get C tasks
  - c. **Loop2:** For every  $up_m^t \subseteq [0,1,2 \dots C]$  tasks
    - i. **Loop3:** For every  $su_n^t \subseteq [0,1,2 \dots N]$  do:
      1. Get the updated cost function  $f(n)$  of allocation tasks  $up_m^t$  with timing requirement  $ut_m^t$  to server  $s_n^t$ .
      2. If the  $su_n^t + up_m^t > max_n^t$  is violated penalize the server that has been over utilized by factor  $K$
      3. Get min where min is the location where the minimum  $f(n)$
    - ii.  $su_{min}^t \leftarrow up_m^t + su_{min}^t$
    - iii. **End Loop3**
  - d. **End Loop2**
  - e. **For all Tasks :**  $ut_m = ut_m - 1/\mu$
  - f. **For all Tasks** If (  $ut_m \leq 0$  ) task m is going to be assumed finished and removed from task queue.
  - g. The unfinished tasks are moved to the  $O$  queue
3. **End Loop1:**Exit simulation

Energy savings in a datacenter using BF allocation are achieved, because BF reduces the number of active servers by consolidating several jobs into the least amount of servers possible. If a server has not received a task from the centralized scheduler, the scheduler switches the server into a low power mode -sleep mode-. A sleeping server consumes much less energy than an active server or an idle one, since the constant power is reduced to a sleeping power. A sleeping server can turn off its

fans and disks and put memory and CPU into a sleep or very low power state which significantly reduces power usage. The SLAV metric was introduced to make sure that even if we are trying to save energy consumption, we must not overcrowd the working servers, since over-utilization would result in slowdowns and sometime system failure. Note that BF works well for homogeneous datacenters as well as heterogeneous ones. The BF algorithm is a brute force method for a given task order, this result in high processing time, especially compared to RR.

This algorithm iterates on all servers and for every available task resulting in a significant energy reduction at the expense of a high computation time. By calculating  $f(n)$  for every task onto every server we can select the one with the lowest cost. FF on the other hand assigns the tasks to the first server having enough computational capability to execute it without checking if it is the best server to do so. A task should be assigned only once. After getting the vector associated with the cost of assigning the task to every possible server, the algorithm chooses the one with the lowest cost. This process is repeated until the queue is empty and every task has been assigned. Once the queue is empty the algorithm waits for the next queue to start task scheduling all over again.

A task can only be considered completed when both processing and timing requirement are met. The task is complete gets removed from the task queue to make room for new tasks. After a  $T$  amount of time has passed, new  $M$  tasks are going to be added to the queue, and will be combined with the non-completed tasks and sent to be scheduled.

In this form the BF does look like a serial problem, because the assignment of task  $m + 1$  depends on the available processing capability to the  $N$  servers which in turn depends on the allocation of the task  $m$ . The algorithm is divided into three loops. The outer loop is responsible for generating the tasks and merging unfinished tasks with the new batch of tasks, the middle loop is responsible for looping on all the

available tasks, and the most inner loop is responsible for selecting the best server for running the current task. The order of the incoming tasks does have an effect on the allocation but not too significant, mainly because in datacenters are weakly heterogeneous and the limited variation in the workload means that pattern will start to appear with tasks.

This variation of BF is ideal to be used in datacenter for two reasons:

- 1- By checking all available location the algorithm can make sure that the tasks assigned give the lowest energy possible.
- 2- By penalizing the over utilized servers or the potentially over utilized, makes sure the application is safe to run while minimizing lag or slowdown.

The big drawback of the algorithm is the high computation required to assign these tasks, and in real world this high execution time would render this method unusable for online application if the arrival rate of tasks are higher than the execution time of the algorithm as shown in later subsection.

To resolve the issue of the high execution time, we propose a two-step BF algorithm that reduces the computational complexity, and allows multicore and many core machines to run it thus resulting in a huge speedups.

Dividing the large problem into a subset of smaller problem is a common approach. However in this case the divide is less obvious since each step of the algorithm depends on both: the currently available tasks and the available processing capability of servers, so the divide should happen for both  $N$  and  $M$  at the same time. Doing this divide ensures that each block forms an independent smaller BPP, and while the computational complexity of the BF is of the order of  $N$  by  $M$ . A divided would have a  $Blk$  smaller Block - where  $Blk$  is the number of blocks- would have a computational complexity of  $N / Blk$  by  $M / Blk - O(N * M / Clt^2)$ - , which is a quadratic reduction in complexity. The second benefit of dividing the large problem

into a smaller independent problem is that it allows us to assign each smaller problem into a different CPU thread which allows even faster execution. So a quad core machine can ideally divide the large BPP into 4 smaller BPP resulting in nearly 16 times speedups. The Speedups using divide and conquer should be almost linear with the number  $Blk$  for a single core device or a single thread device and quadratic for a device having  $Blk$  numbers of cores. If the Problem is divided into more blocks than the number of cores available for the system, the speedups would be linear.

However, there is a limit on how many blocks can be divided. First, the CPU can only work limited amount of threads. Having many threads would result in many context switches and less performance and sometimes crashing. Second, dividing the data and creating threads is not a free process, it requires some overhead, and the number of threads should be chosen carefully to ensure speedups. Third, dividing large problem into smaller ones means that each will give a local solution rather than a global solution. However, when using a small number of blocks, and large number of servers and tasks, the speedups are going to be very high, and the local solution should be very close if the relative number of tasks and servers is very large to the number of blocks since the waste is going to be minimal.

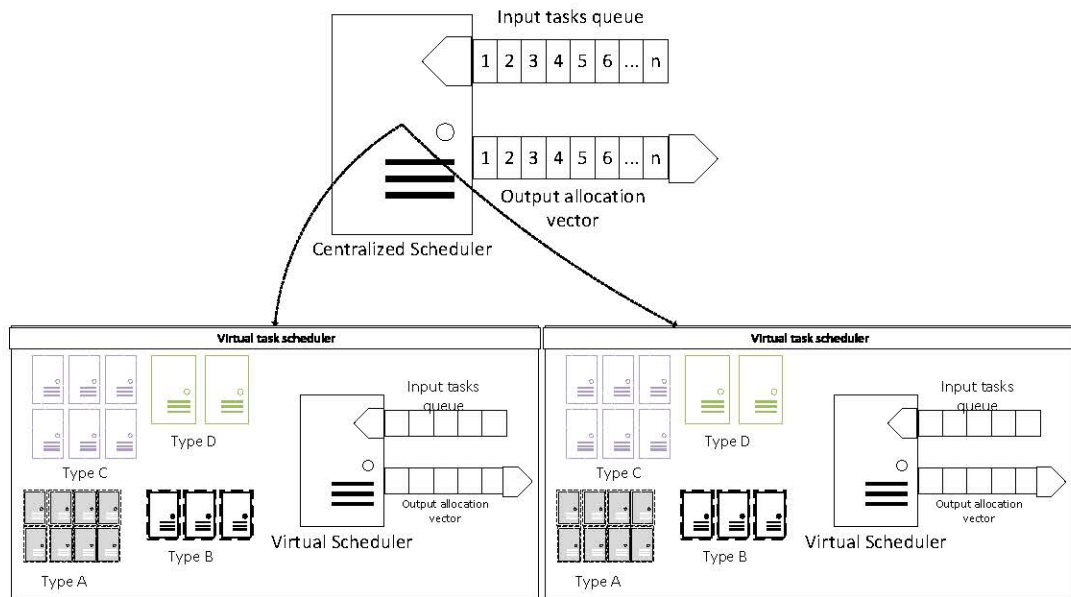
### **3.6 DIVIDE AND CONQUER BEST FIT**

The first proposed solution step is DCBF, this step is not OpenMP or CUDA specific, and so it would work on practically any language, even non multicore programming language, and it goes as follows. Instead of assuming the BF would be done on entire datacenter, we can dynamically divide the datacenter into virtual blocks and optimize the scheduling on a local level, so instead of having  $N$  tasks loop through  $M$  Servers, we can have  $Nblock$  tasks loop through  $Mblock$  servers, where  $M$  is total number of tasks and  $N$  is the total number of servers,  $Nblock = N / Blk$

which represents the total number of servers in a block and  $Mblock = M / Blk$  which is the total number of tasks in a block  $Mblock$ ,  $Blk$  is the number of blocks.

It is important to divide the number of tasks equally in all clusters or blocks or we might end up with some clusters be overloaded and other underutilized. Using cluster division is reasonable since in real datacenters servers are clustered based on their geographical location. The speed and the performance of DCBF approach depends on the size of  $Blk$ . In DCFS, BF is just a special case of DCBF where  $Blk$  is equal to 1. Selecting an appropriate  $Blk$  is essential as a too low selection would not yield enough speed improvement, while too high  $Blk$  would not give too much energy reduction and might also give some improvement over Round Robin in term of SLAV.

Figure 8 shows the overview of DCFB, where the centralized scheduler virtually divides the datacenters into two local scheduler, since this divide is a virtual divide it can be dynamic with minimal overhead. This divide allows multiple threading to happen and have a significant impact on the speed of the algorithm.



**Figure 8: Virtual Local Schedulers**

The traditional BF has a computational complexity of  $O(N * M)$ , while in DCBF each block have a computational complexity  $O(N * M / Blk^2)$ . For a single core machine DCBF computation time is proportional with  $N * M / Blk$ . Since all clusters would be optimized locally, it is easier to take advantage of parallelism since each thread can work on a local optimum independently. For multicore systems, we can split every cluster into a different working threads as each subset is independent which results in a significant speedups. At best case the execution time is reduced by a factor of  $Blk^2$  (for multicore system that can handle all available threads simultaneously), and reduced to a factor of  $Blk$ , for a core single thread system. Choosing the correct value of  $Blk$  is essential, as a high  $Blk$  can result in too many CPU threads, in which the benefit of parallelism is overshadowed by the overhead required to fork and join the threads for multicore systems. The result might even be worse on single core systems due to the many context switches required.

The algorithm goes as follows:

1. Initialize values ( $N, M, R, Blk, K, \dots$ )
2. **Loop1** for number of iteration  $I$ 
  - a. Generate  $M$  number of tasks based on random number generators
  - b. Combine  $M$  tasks with previously unfinished  $O$  tasks and get  $C$  tasks
  - c. Divide  $N$  tasks and  $M$  servers into  $Blk$  cluster and  $Blk$  threads
  - d. **Loop2:** For every  $up_m^t \subseteq [0,1,2 \dots M/Blk]$  tasks in every thread
    - i. **Loop3:** For every  $su_n^t \subseteq [0,1,2 \dots N/Blk]$  in each thread:
      1. Get the updated cost function of allocation tasks  $u_{pm}^t$  with timing requirement  $ut_m^t$  to server  $su_n^t$ .
      2. If the  $su_n^t + up_m^t > \max_n^t$  is violated penalize the server that has been over utilized by factor  $K$

3. Get min where min is the location where the minimum energy difference happened
  4.  $su_{min}^t \leftarrow up_m^t + su_{min}^t$
  5. **End Loop3**
- ii. **End Loop2**
- e. Join forked threads
  - f. **For all Tasks** :  $ut_m = ut_m - 1/\mu$
  - g. **For all Tasks** If (  $ut_m \leq 0$  ) task  $m$  is going to be assumed finished and removed from task queue. The unfinished tasks are moved to the  $O$  queue
  - h. **End Loop1**
3. Exit simulation

The speed ups using this method should be significant. It can be implemented with several programming directives using a programming model like OpenMP, MPI, OpenCL, CUDA ...

The algorithm described above favors CPU execution, since there is still a large serial part. The big and obvious improvement in DCBF is having a smaller number of computations due to a smaller number of servers and number of tasks per block. The other subtle improvement is the better utilization of the CPU cache - a smaller array uses less cache-, hence reducing the number of accesses required to the main memory. A reduced access to memory can give a large improvement in performance.

While this method gives a significant speedup boost, it is not enough for strict timing requirements. Furthermore, it is not optimized for the GPU. The GPU has a smaller cache and a larger number of cores, and can handle much more threads than the CPU counterpart. To further improve the allocation technique, we propose optimizing it for GPU, by trying to reduce the serial part and allow more simultaneous threads .



Using BF as a task scheduler should provide good energy reduction compared to the traditional Round Robin at the expense of high compute time. Using DCBF should reduce BF execution time by a decent amount with minimal hit on energy reduction. However, when  $N$  and  $M$  are both high, a decent improvement is not enough, so we looked into more optimization techniques to reduce compute time even further. Thus, we propose a GPU aided BF algorithm that we call Accelerated Best Fit.

### 3.7 ACCELERATED BEST FIT

ABF is a modified version of BF and DCBF that taps into the raw computational capability of the GPU to give a huge speed increase. The improvement provided using ABF should be so significant that it should transform BF from being an offline BPP solution into an online one. ABF uses the advantages of massive threading capability of the GPU, along with shared memory and thread cooperation to achieve the desired speedups. But first we must expose where DCFB can be parallelized and how we can take advantage of thread cooperation to reduce the computational complexity compared to BF and DCBF.

#### 3.7.1 Radix Min Position

The Radix Min position method is a proposed method to get the position of the minimum with a shared memory environment without altering the original array. This step is necessary in CUDA to reduce the complexity of the most inner loop. The Radix Min Position is implemented as follows:

1. Generate an array  $pos$  with size  $S$
2.  $S \leftarrow S/2$
3. For every thread with index  $i, (i < S)$ 
  - a. Compare  $f(pos(i))$  and  $f(pos(i + S))$

- b. The position that gives the lower result is copied to  $i$
- c. Synchronize threads ( to prevent data bank conflict)
- d.  $S \leftarrow S/2$

4. The smallest position is found at index 0

The following method manages to reduce the number of iteration of finding the index of minimum from  $N$  steps to  $\log_2 N$ . This is a substantial reduction in computational complexity. Add that to an intelligent use of shared memory and this method ensures much faster results than when using a serial method to find the minimum. Radix Min Position should in theory work on any system with shared memory and thread cooperation but it is best performed using the GPU. The only downside of this method is that it requires double amount of memory than when executing the code serially. The downside is important since the amount of shared memory on the GPU is limited.

### ***3.7.2 CUDA optimized Best Fit algorithm***

While DCBF is a very good performance leap, it is more optimized toward multi core processors rather than GPU. In this thesis we optimized ABF to run much faster on the GPU by using what the GPU and CUDA has to offer.

Unlike CPU threads which require thousands of cycle and create large overhead, a GPU thread is light with very minimal overhead. Therefore launching thousands of thread or even hundreds of thousands of threads is feasible on the GPU, it is unfeasible on the CPU. The second advantage that the GPU has over the CPU is the control the programmer have over the shared memory. While the CPU has larger cache, it is not controlled by the programmer but handled dynamically by the CPU, which makes finding Radix Min Position impossible on the CPU. The GPU cache or shared memory has two benefit over the CPU cache that helps dramatically improve the performance. Shared memory as the name suggest is shared at CUDA block level, which means all threads within the same block can cooperate on completing a task,

such as when using Radix Min Position. These algorithm can reduce the computational complexity from  $N$  to  $\log_2(N)$ . Second, the performance benefit for shared memory is hiding some memory latency by managing copying memory from the GPU main memory to the shared memory. In other words while some threads are being executed as warps, others are getting data from the memory. For best performance and error free execution, the global memory access should be coalesced, which means that sequential threads should access sequential data locations form memory, this would help takes advantage of the very high bandwidth rate of the graphics memory available for the GPU. Since the memory is shared in the cache we must make sure of no access violation incurring inside, to do so CUDA provides some functionality to enable this sort of management, such atomic operations, mutex locks and thread synchronization. These add some additional difficulties for the programmer to handle.

ABF differs from DCBF to allow faster execution on the GPU with the following regards:

1. Instead of dividing each cluster to a single thread, we can divide each cluster into a block of threads, these threads should be sized in the power of 2. A common size for a block is 32, 64, 128, 256, 512 or 1024 threads per block. Older GPU however are limited to a maximum of 512 threads per block such as “Tesla” GPU. Newer GPU can handle up 1024 threads per Block. If the servers and tasks number is not equal to the power of two, it is necessary to pad the task and server information with virtual tasks and servers to the next power of 2. Because many parallel algorithms such as Radix Min Position requires the number of elements to be a power of two.
2. Instead of having one thread calculating  $f(n)$  for all servers in BF and DCBF, we can have each one of the GPU threads calculating  $f(n)$  for one server.

3. Using shared memory we can reduce getting the minimum position from  $O(N)$  to  $O(\log_2(N))$ , it is done by allowing threads to cooperate to get the minimum. The shared memory is also used to reduce the number of access to the main memory, thus greatly reducing the wait time to get the task data.

However each one of these improvements present a challenge on its own. The limited availability of shared memory means that the size of shared memory used should be kept at minimum, this can be done by reusing the shared memory when possible. Another difficulty is ensuring that there would be no data bank conflict: this is difficult since each thread can be in a different point of execution in the code, hence the use of thread synchronization and atomic operations. Copying data from the GPU main memory to the shared memory should be done in a single operation, thus reducing latency and preventing overloading of the GPU memory. Also the data copy should be made in coalesced manner. Repeated memory request can cause errors in the memory. Finally, when the number of servers or tasks are not a power of two, the data have to be padded –by adding zero-to the next power of two to make sure that the radix min position gives correct results.

The GPU optimized ABF follows this process:

1. Initialize values ( $N, M, R, Blk \dots$ )
2. **Loop1** for number of iteration
  - a. Generate  $M$  number of tasks based on random number generators
  - b. Combine  $M$  tasks with previously unfinished  $O$  tasks and get  $C$  tasks
  - c. Compare  $N$  and  $C$
  - d. Pad the highest of the two to the next power of two
  - e. Divide  $C$  tasks and  $N$  servers into  $Blk$  cluster and the highest power of 2 threads
  - f. Copy data to the GPU

g. **Loop2** :For every  $up_m^t \subseteq [0,1,2 \dots C/Blk]$  tasks in every CUDA Block

i. Assume every thread calculate the cost function and the server utilization  $su_n^t \subseteq [0,1,2 \dots C/Blk]$  :

1. Get the updated cost function  $f(n)$  of allocation tasks  $up_m^t$  with timing requirement  $ut_m^t$  to server  $su_n^t$ .
2. If the  $su_n^t + up_m^t > sumax_n^t$  is violated penalize the server that has been over utilized by factor K.
3. Use Radix Min position method to get the minimum.

ii.  $s_{min}^t \leftarrow ut_m^t + s_{min}^t$

iii. **End Loop2**

h. Copy allocation vector back to the CPU

i. **For all Tasks** :  $ut_m = ut_m - 1/\mu$

j. **For all Tasks** If (  $ut_m \leq 0$  ) task m is going to be assumed finished and removed from task queue. The unfinished tasks are moved to the O queue

k. **End loop1**

3. Exit simulation

BF and DCBF requires high Computational Complexity of  $O(N * M)$  and  $O(N * M/Blk^2)$ . While the GPU optimized algorithm has a much lower computational complexity of linearithmic time  $O(N * \log_2(M)/Blk^2)$ . In addition computing  $f(n)$  on the CPU requires a linear time  $O(N)$ , the same calculation can be done in CUDA in constant time  $O(1)$ . Going from a quadratic complexity to a linearithmic times is a huge reduction in complexity and should provide significant speed increase on top of DCBF. On the other hand the reduction in energy

consumption and SLAV minimization should be slightly worse in ABF than in DCBF or BF due to the reduced size of the block.

### 3.8 SCALABILITY

In queuing theory terms one can consider the BF scheduler as a M/M/1 or D/D/1 queue, and the proposed DCBF and ABF falls under M/M/Blk and D/D/Blk queues. In all of these cases, the assumptions that all queues are infinite. However in reality an infinite queue is not a realistic.

The datacenter scheduler is getting tasks in its queue at rate of  $M$  tasks per batch at every  $1/\mu$  second, with average processing requirement  $UP$  and Wait Time  $UT$ , the total Processing Unit Required per Batch Should be  $\sum_0^m up_m^t * ut_m^t$ . Since  $UP$  and  $UT$  are independent, the expected value of  $\sum_0^m up_m^t * ut_m^t = M * UP_{avg} * UT_{avg}$

The total number of servers  $N$ .  $n_1, n_2, n_3 \dots n_n$  each  $n$  represent a server. Each type of servers has its own power model. Assuming that each type has a maximum processing capability available  $sumax_1, sumax_2 \dots sumax_n$ , and the average server processing capability is  $SU_{avg}$ . The available processing capability available at every period in the datacenter is  $\sum_0^n sumax_n/\mu$ . Assuming that  $N$  and  $sumax$  are independent the expected value of  $\sum_0^n sumax_n/\mu = N * SU_{avg}/\mu$ . As long as the available processing capability is more or equal to the required processing capability the servers shouldn't be over utilized. The same logic can be applied for both BF and to DCBF as well as ABF, as long as we are dividing the tasks and servers equally into clusters.

$$M * UP_{avg} * UT_{avg} \leq N * SU_{avg}/\mu$$

If this inequality is violated we can expect an infinite queue to build up which prohibits the datacenter from functioning normally and can cause a service downtime. Dependent on the application, a downtime can have some serious ramification especially in banking.

Since  $U_{avg}$ ,  $P_{avg}$  and  $S_{avg}$  are constant if the random number generator is consistent, the algorithm should give scale dependent on the arrival rate of tasks and the number of tasks per batch and the average processing capability of the datacenter by the following relation:

$$N \propto \frac{M * \mu}{S_{avg}} \quad (10)$$

$$N \geq \frac{UP_{avg} * UT_{avg} * M * \mu}{SU_{avg}} \quad (11)$$

## CHAPTER 4

### SETUP AND TESTING

To test the validity of the algorithm we developed a simulator that runs these schedulers, and ran it on two different machines. The first is a 14" inch laptop -MSI GE40- with the following specifications:

- Intel i7-4702mq quad core CPU 2.2 GHz with 6MB cache
- 16 GB of 1600 DD3L RAM dual memory channel
- Nvidia GTX 760m with 2GB of GDDR5 memory, with optimus technology
- 128 GB of SSD + 750 GB of 7200 RPM HDD
- Windows 8.1 64 bit professional
- CUDA toolkit 6.5
- Visual studio 2012

This test setup is our development and host machine. It is a fairly regular machine with high end Mobile CPU and a mid-range gaming class GPU. The Nvidia GTX 760m is "Kepler" generation GPU with Compute capability 3.5, 768 CUDA cores with 4 SM with 192 CUDA cores, the clock speed is rated at 657 MHz. The latest drivers were used in the run. The code is compiled using NVCC compiler in release mode and x86 target platform under Windows. Using a higher end or a newer GPU can yield much faster results, note that year over year releases, the GPU are rapidly growing in performance while CPU performances are begging to stagnate. At the time of writing the GPU performance increase per generation is around 40%, while the CPU performance increase per generation is about 10%. Beside that GPU generations are faster to release at a rate of almost twice a year while the CPU generation are released at a rate of once a year, making the GPU a lucrative choice for porting algorithms. Beside that GPU upgrades in a desktop or a server is much easier



than a CPU upgrade that often requires an upgrade to the motherboard and other component as well.

The Second system is an aging desktop computer accelerated by modern GPU to provide fast algorithm processing. The machine specifications are as follows:

- Intel e6600 2.4 GHz core 2 duo desktop CPU with 4 MB cache
- 4 GB DDR2 rams
- Nvidia GT 740 with 2gb GDDR5 rams with 384 CUDA cores and 2 SM
- windows 7 32 bit ultimate
- 180 GB HDD with 5400 RPM

System 2 should show the improvement resultant by using a budget modern GPU, note that modern computers support PCIe 3.0 while old computers are limited to PCIe 1.0 which limit the communication speed between the CPU and the GPU. This bottleneck might theoretically reduce the algorithm performance. The bottleneck is due to PCIe working at the lowest speed of the components communicating on it for backward compatibility. PCIe 3.0 is 4 times the speed of PCIe 1.0 for the same number of data lines.

In this thesis, we are assuming that the servers are weakly heterogenous, which means that all servers fall under one of few pre-determined configuration. This is reasonable for most data centers.

In our implementation, we assumed that there are 4 types of servers. Table 1 shows the unique characteristics of each type of servers.

Server Type	$\alpha$	$\beta$	<i>idle</i>	<i>sumax</i>
Type A	6	1.05	2	20
Type B	60	1.4	3	60
Type C	76.8	1.3	4	100
Type D	79	1.2	5	200

**Table 1: The characteristics of the four types of servers considered.**

Type A servers are represents low power low performance servers, such as an ARM server or an Intel Atom server. These are commonly used as an I/O servers. Type B servers represent low end desktop server such as Celeron or AMD Athlon server. Type C servers represents a single CPU high end server with a single CPU. Finally, type D servers represents high end servers that can have two CPU per board. These are expensive high end servers reserved for running demanding tasks.

Each run in the simulations is repeated 20 times for each configuration  $(M, N, \mu)$ , to prove the algorithm scaling under different loads. *Blk* is chosen at run time and it is equal to the number of concurrent threads that the CPU can run to get the best efficiency from the hardware. The CUDA block size is varied between (32, 64, 128, 256, and 512) showing the effect of block size on the speed of the algorithm and the effect on energy reduction and SLAV. The total execution time is summed for each block size independently for a total of 20 periods. Each run begins slowly when the queue is empty and starts to fill up rapidly until it gets to a steady state. This will allow us to compare the speed of the algorithm along with the power reduction and SLAV for each run. Each run is repeated 4 times, and the results is the average of the 4 runs to prove the consistency of the run. Hence, a total of 1600 scheduling runs for each test system. The large number of runs should prove that the result is consistent, scalable and stable.

Assuming that  $N=20000$  servers,  $M= 2000$  tasks per batch and  $T=1/\mu = 10$  seconds.

## 4.1 RESULTS AND DISCUSSION

Figure 9 shows the total energy in datacenters using the proposed algorithms.

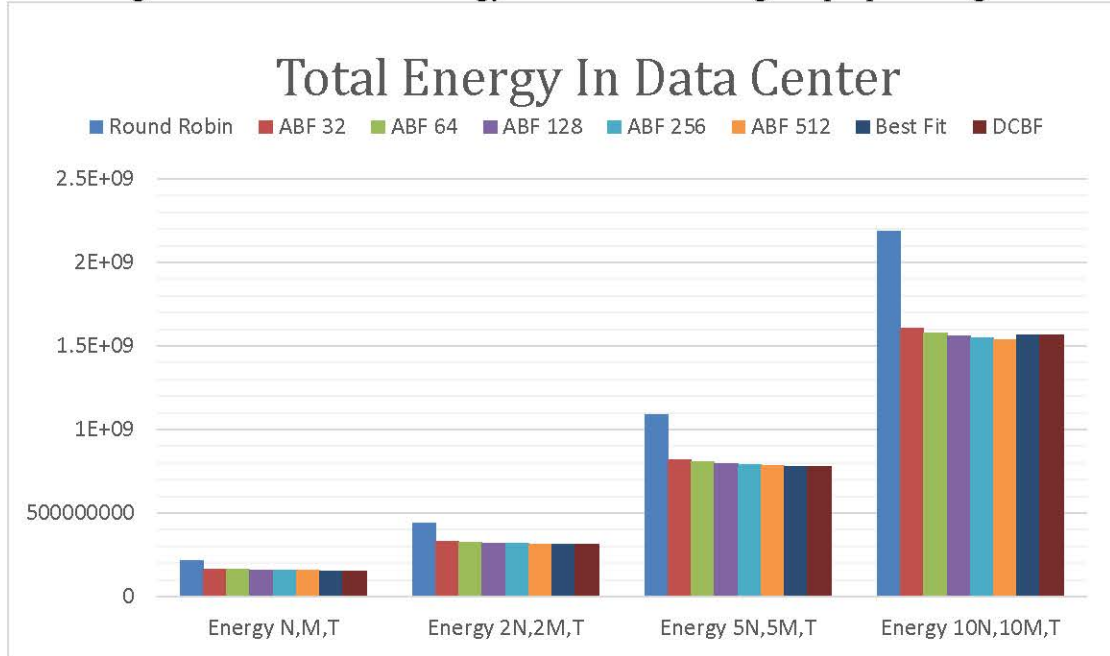


Figure 9: Total Energy in Data Centers

ABF 32 represents the run where the datacenters is virtually divided into groups of 32 servers and the scheduling is ran locally in each block, ABF 64 into groups of 64 and so on. BF optimize task scheduling for the entire datacenter. DCFB on the other hand virtually divide the datacenter into a small number of blocks equal to the number of concurrent threads which the system can run, which is equal to 8 blocks in system 1. The energy saving using BF, DCBF and ABF in datacenters varies between 24% energy reductions while using ABF 32 to 28% while using BF compared to RR. The difference in energy saving between ABF 512 to BF or DCBF is less than one percent, hence the hit on energy reduction is very small.

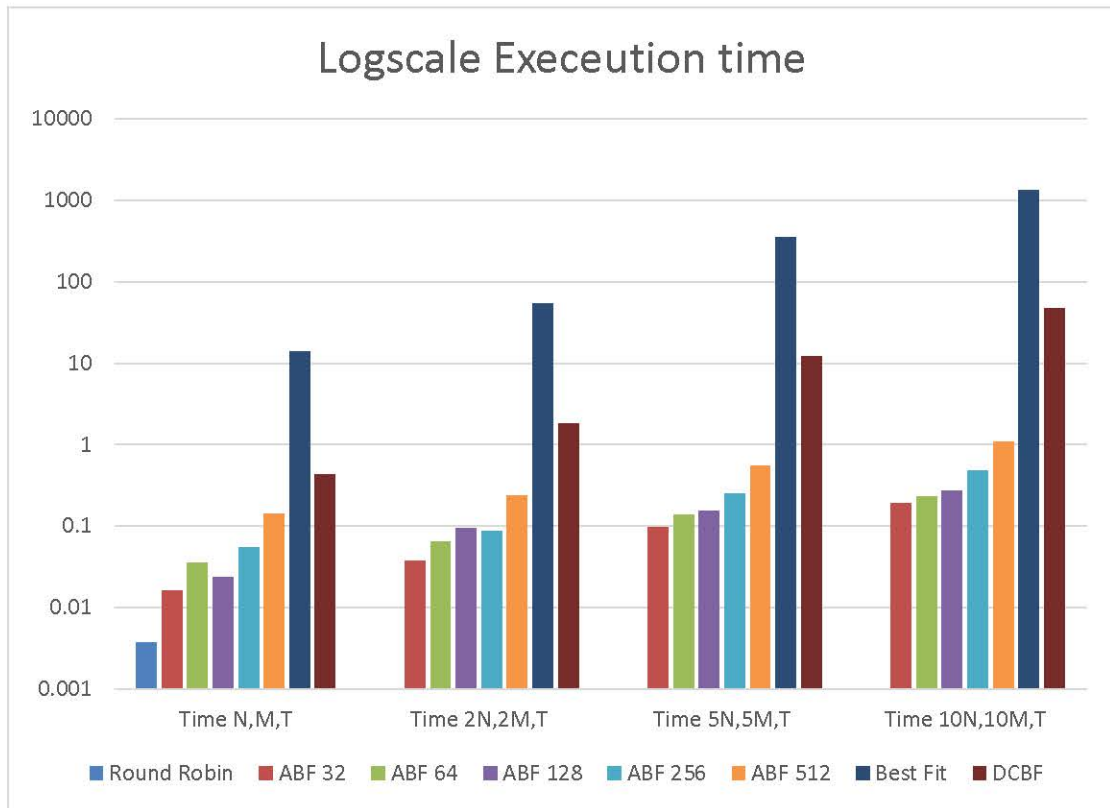
In the first batch of simulations  $N$  is originally assumed to be 20,000 and  $M$  2,000 and  $T$  fixed at 10,  $N$  and  $M$  where varied in a linear manner to maintain the ratio  $N/M * T$ . Figure 9 shows the energy saving using BF, DCBF and ABF are

significant. But since it is known that the BF algorithm is computationally expensive, the time result are shown in figure 10.

	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
E reduction N,M,T	0	24.20091	25.91324	26.94064	27.62557	27.85388	28.76712	28.65297
E reduction 2N,2M,T	0	24.17332	25.82668	26.96693	27.53706	27.87913	28.79133	28.67731
E reduction 5N,5M,T	0	24.73684	26.17849	27.1167	27.62014	27.98627	28.627	28.60412
E reduction 10N,10M,T	0	26.48402	27.85388	28.76712	29.22374	29.68037	28.53881	28.53881

**Table 2: Energy reduction percentile compared to round robin**

The time difference in execution is huge. Theoretically, BF execution time is quadratic with respect to the number of tasks and servers. In other words when the number of tasks and the number of the servers double, the execution time quadruple. This can be clearly seen in figure 11. The time required for running BF is too large for online application when the number of servers and tasks are both large. For example,  $10N$  and  $10M$  needed 3.78 more processing time to performe than  $5N,5M$ , since both  $M$  and  $N$  are doubled



**Figure 10: Logscale execution time**

Figure 10 shows the time difference in execution between different algorithm, the speedups are so significant that it can't be easily seen in a linear figure hence the use of logscale axis. Table 3 shows the relative speedups obtained of using DCBF and ABF compared to BF. RR execution time number are also included. For DCBF the time of execution include dividing the datacenter, forking, executing local scheduling and then joining the threads to get the global allocation vector. ABF execution time include dividing the datacenter, sending the tasks data to the GPU, launching the massive number of threads, running the algorithm, getting the local optimum results and finally copy the allocation vector back to the CPU. This means the actual ABF and DCBF speedups are even larger then what it appears, but including the communication and threading overhead for more practical purpose.

Speedups	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
Time N,M	3751.333 333	879.2188	401.9286	598.617	254.6154	99.94671	1	32.63921
Time 2N,2M	infinite	1456.149	825.7088	565.643	621.0663	224.9582	1	29.34504
Time 5N,5M	infinite	3649.378	2605.749	2352.888	1437.745	657.1226	1	29.27568
Time 10N,10M	infinite	6986.331	5801.953	5020.817	2807.591	1244.802	1	28.81053

Table 3: Times Speedups compared to BF

As seen in table 3 DCBF is getting around 29 times speedups compared to BF, this speedups is the results of two things. The first reason is the reduced complexity because the set is 8 times smaller which should give around 8 times less work. The second reason is that taking advantages of the multicore capability of the CPU which should give around 4 times speedups since system 1 is a quad core machine. DCBF can be done without multicore system, while Multicore systems can't cooperate to do BF. Table 3 proves that the real speedups (~29 times) using DCBF is very close to the theoretical one (~32 times). While in ABF, the speedups are even more significant depending on the size of the of the blocks,  $N$  and  $M$ . The speed increase ranges from 100 times speedups to a nearly 7000 times even when taking into consideration the overhead of CPU/GPU communication back and forth, preprocessing and post processing overhead of virtually dividing the datacenter into blocks. The actual algorithm time without the overhead was measured to is as low  $2 \mu s$  using CUDA profiler, which means an improvement can be in the order of millions if the communication overhead can be reduced to zero.

Even further improvement can be attained using a more powerful GPU. The GPU used in the previous simulation is single midrange notebook GPU, which can operate a maximum of 768 threads at once. During simulations the number of GPU

threads can reach up to  $2^8$  threads which equal 262144 threads. Having a more powerful GPU or multiple GPUs can help reduce the compute time even further. At the time of the writing the most powerful GPU in the market –Nvidia Titan Z- having 8122 GFLOPS [39]. System 1 GPU has 964.6 GFLOPS [40] of compute performance. The other benefit of using GPU for running the algorithm is that it frees the CPU to work on other jobs while the GPU is busy getting the optimal allocation vector.

	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
SLAV N,M	17475000	0	0	0	0	0	0	0
SLAV 2N,2M	35050000	0	0	0	0	0	0	0
SLAV 5N,5M	87700000	0	0	0	0	0	0	0
SLAV 10N,10M	175000000	0	0	0	0	0	0	0

Table 4: SLAV for fixed T and variable N and M

Table 4 represents the SLAV which results from overutilization of servers. In this configuration the SLAV only incurs while using RR, which means that some computationally heavy tasks were assigned into low power servers that is not optimized to run it. BF, DCBF and ABF avoid over utilization by placing a demanding task into powerful servers and non-demanding tasks into low power one to reduce energy consumption.

Figure (9-10) and table (2-4) proves the scaling of the algorithm regarding the number of servers and the number of tasks. To prove the scaling with respect to the arrival rate,  $N$  is fixed at 20000,  $M$  and  $T$  are varied with the base value of 2000 and 10 respectively.

In figure 11 the algorithm keeps the shape consistency, with RR consuming the most energy, followed by ABF with smaller blocks. Largest blocks have the lowest energy consumption hence largest energy saving.

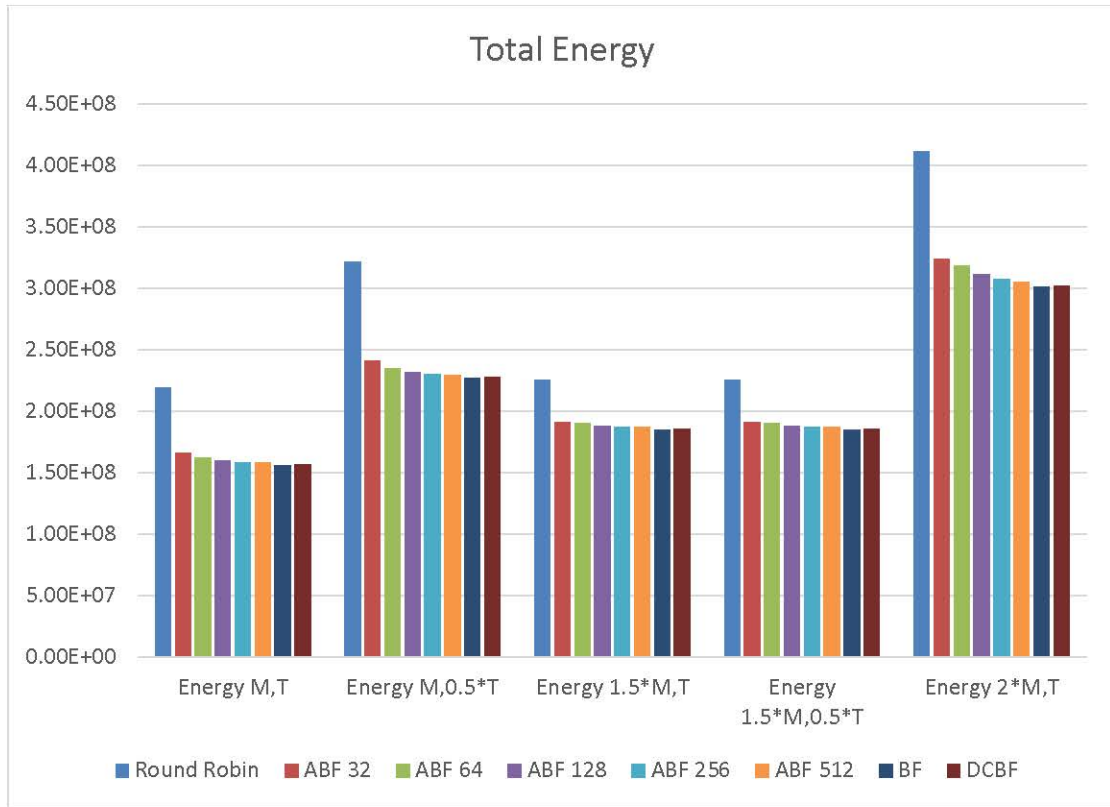


Figure 11: Total energy for fixed N and variable M and T

	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
E reduction M,T	0	24.20091	25.91324	26.94064	27.62557	27.85388	28.76712	28.65297
E reduction M,0.5*T	0	25.05837	26.84825	27.93774	28.48249	28.71595	29.2607	29.18288
E reduction 1.5*M,T	0	15.11111	15.55556	16.44444	16.88889	16.88889	17.77778	17.55556
E reduction 1.5*M,0.5*T	0	15.11111	15.55556	16.44444	16.88889	16.88889	17.77778	17.55556
E reduction 2*M,T	0	21.20292	22.661	24.36209	25.33414	25.82017	26.79222	26.54921

Table 5: Energy Reduction Percentile

Table 5 shows that energy reduction percentile for BF, DCBF and ABF compared to RR when  $N$  is fixed while varying  $M$  and  $T$ . This prove that the algorithm is scaling correctly in all dimensions ( $M$ ,  $N$  and  $T$ ). As the datacenter becomes more cluttered with tasks, the power reduction using BF, ABF and DCBF are less significant (29% to 17.5%).



	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
SLAV M,T	1.75E+07	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
SLAV M,0.5*T	14575000	118370.8	5475.75	0	0	0	0	0
SLAV 1.5*M,T	26275000	5591.5	0	0	0	0	0	0
SLAV 1.5*M,0.5*T	33075000	11450000	10925000	9427500	6837500	7567500	1.03E+08	19550000
SLAV 2*M,T	40850000	858743.3	47667.5	0	0	0	0	0

Table 6: SLAV

Table 6 shows the SLAV under different workload condition density. If either the batch arrival rate or the tasks per batch doubles, SLAV might start to appear. Which means not only power reduction has maintained its percentile, also the SLAV has maintained its percentile with different load conditions and different sizes. SLAV was violated in all servers under extreme workload conditions.

In all of the previous tests the result are the average of four runs. The standard deviation for energy does not exceed 1%, while the standard deviation of time of execution varies between 3% and 13%. This is an additional proof of the consistency of the algorithms.

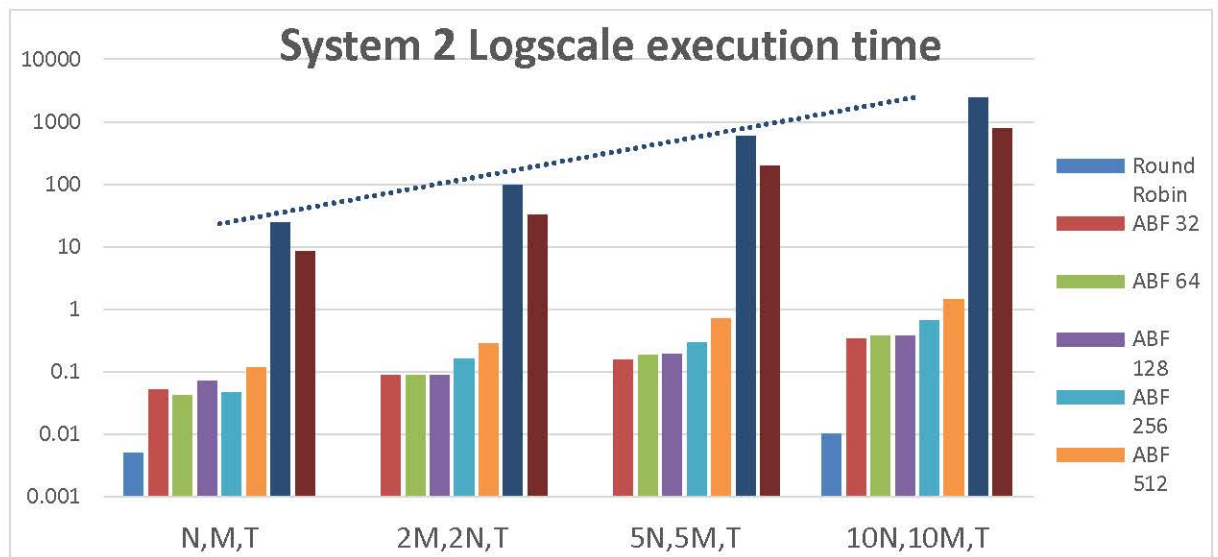


Figure 11: System 2 logscale execution time

Figure 12 shows the execution time of the algorithms running on system 2. This system has a much older CPU and weaker GPU. The GT 740 has a compute capability of 762.6 GFLOPS [41]. Which makes it around 20% weaker than the GPU used in system 1

	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
N,M,T	4891.533	467.3439	577.7402	338.1244	516.7113	206.6845	1	2.887678
2M,2N,T	Infinite	1078.283	1098.705	1082.306	598.0577	338.0629	1	3.014153
5N,5M,T	infinite	3811.92	3196.876	3082.116	2070.078	850.4896	1	3.017601
10N,10M,T	232966.1	7186.02	6396.767	6494.559	3665.964	1661.746	1	3.044463

Table 7: The relative performance increase compared to BF

A huge relative performance increase can be seen in table 7. ABF relative speedups are very similar to that seen in table 2 (system 1), however the DCBF relative speedups is much lower -3 times system 2 compared to 30 times of system 1-. This difference is due to two reasons. The first is the number of blocks, DCBF will dynamically divide the work into 2 blocks in system 2 while it is divided to 8 blocks in system 1. Thus resulting in system 2 having 4 times more work to do than system 1. The second reason is that system 2 can only process two concurrent threads while system 1 can process 8 concurrent threads.

SLAV and the total energy results using system 2 is nearly identical to system 1, since both ABF and DCBF behave almost the same under different machine and different configuration. The only major difference is their speed of execution on different hardware.

Not only is the algorithm capable of doing the same tasks in much lower time. Just by having a more powerful GPU we can see a huge difference in performance. And given that the generational increase in GPU performance is around 40%, and the CPU performance increase per generation is around 10% makes ABF future-proof. Besides that upgrading a GPU is as simple as removing the old PCIe GPU card and replace it with a newer PCIe GPU card. A CPU upgrade most likely requires an

upgrade to the motherboard and sometimes RAMs which might dramatically increase the cost of upgrade. In our setup, installing the modern GT 740 GPU didn't require any changes to the now aging system –about 8 years old-, just inserting the new graphics card and installing its drivers. A CPU upgrade most likely requires an upgrade to the motherboard and other components which dramatically increases the cost of an upgrade. Despite all that it still wouldn't give the same speedups as using ABF.

$1/\mu$	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
7	16060625	132	0	0	0	0	0	0
6	15257475	4285.75	0	0	0	0	0	0
5	14444025	35006.75	526.75	0	0	0	0	0
4	14562750	153484.3	27679.75	1897.25	0	0	0	0
3	13303725	362418.8	165637.8	63727.75	15971.5	4817	0	0
2	10637525	2.73E+08	1.24E+10	1.24E+10	1.24E+10	1.24E+10	1.08E+08	17307850
1	6199158	839420	453727.5	331377	242154	184618.3	4.95E+08	67544800

**Table 8: SLAV vs arrival rate for N=20000, M=2000**

Table 8 shows the variation of the SLAV with the arrival rate for  $N = 20000$  and  $M = 2000$  when  $\mu$  is varied. The result obtained is the average of four runs under different batch periods. RR had large SLAV in all runs. As the arrival rate increased, SLAV appeared first in smaller ABF blocks, later SLAV appeared in larger ABF blocks. The ABF based allocation algorithm began to have some SLAV when the ABF with smallest block size of 32 at  $1/\mu = 7$ . Similarity SLAV appeared in block size 64 when the batch period is 5. ABF 128 had some SLAV when the batch period is 4 and below. For both 256 and 512 block size, SLAV incurred when the batch period was equal to 3. The smaller the block size the larger the SLAV incurred. Finally, DCBF and BF incurred some SLAV when the batch period is equal or less than 2.

1/ $\mu$	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
8	3.34E+07	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
7	3.21E+07	1.66E+02	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
6	3.06E+07	8.88E+03	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
5	2.90E+07	6.51E+04	1.55E+03	0.00E+00	0.00E+00	0.00E+00	0.00E+00	0.00E+00
4	2.92E+07	3.25E+05	6.37E+04	5.81E+03	8.00E+00	0.00E+00	0.00E+00	0.00E+00
3	2.66E+07	7.04E+05	3.17E+05	1.21E+05	2.77E+04	7.46E+03	0.00E+00	0.00E+00
2	2.12E+07	1.56E+10	6.90E+10	1.07E+11	6.69E+10	6.71E+10	4.15E+08	6.13E+07
1	1.24E+07	1.69E+06	9.18E+05	6.75E+05	4.93E+05	3.75E+05	2.00E+09	2.70E+08

Table 9: SLAV vs batch period for  $N=40000$ ,  $M=4000$

Table 9 represents the SLAV in the datacenter when  $N=40000$  and  $M=4000$  when  $\mu$  is varied. The result in table 9 are very close to that of table 8 in which the SLAV incurred first at the smallest block size and gradually incurred on the larger block size as the batch period is getting smaller. Finally, SLAV appear when the block is very large or the entire problem is a single block. The incurrences of the SLAV in table 9 is not only gradual as table 8 but also very similar at the batch period that the SLAV began incurring. This proves the scalability of the algorithm, since the ratio of  $N/M$  is maintained.

1/ $\mu$	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBD
7	80317425	0	0	0	0	0	0	0
6	76221800	359	0	0	0	0	0	0
5	72278525	18060	197.5	0	0	0	0	0
4	72626700	253914.3	41697	2772.5	0	0	0	0
3	66742350	5.56E+12	5.56E+12	5.56E+12	5.56E+12	5.56E+12	0	0
2	53032700	6.44E+12	6.44E+12	6.44E+12	6.44E+12	6.44E+12	2.47E+09	3.48E+08
1	30912250	4.35E+12	4.35E+12	4.35E+12	4.35E+12	4.35E+12	1.17E+10	1.55E+09

Table 10: SLAV vs batch period for  $N=100000$ ,  $M=10000$

1/ $\mu$	Round Robin	ABF 32	ABF 64	ABF 128	ABF 256	ABF 512	BF	DCBF
7	1.61E+08	0	0	0	0	0	0	0
6	1.53E+08	862.75	0	0	0	0	0	0
5	1.44E+08	34975	361.75	0	0	0	0	0
4	1.45E+08	478757. 8	74363	4438.5	0	0	0	0
3	1.33E+08	2.22E+ 13	2.22E+ 13	2.22E+ 13	2.22E+ 13	2.22E+ 13	0	0
2	1.06E+08	2.58E+ 13	2.58E+ 13	2.58E+ 13	2.58E+ 13	2.58E+ 13	9.83E+ 09	1.33E+ 09
1	61967200	1.74E+ 13	1.74E+ 13	1.74E+ 13	1.74E+ 13	1.74E+ 13	4.72E+ 10	6.16E+ 09

**Table 11: SLAV vs batch period for N=200000, M=20000**

Similarity table 10 and 11 shows the SLAV results when  $N = 200000$ , 100000 and  $M = 20000, 10000$  respectively. The results are largely similar to that table 8 and 9 since the ration of  $N/M$  is the maintained.

## CHAPTER 5

### CONCLUSION

In this thesis, we introduced AFB and DCBF to solve BPP or VCSBBP for optimal workload placement in datacenters. BF, ABF and DCFB all managed to get around 24~28% percent energy reduction in Datacenter compared to RR while eliminating any SLAV under normal workloads condition. 30 times speed increase was achieved using DCFB compared to BF, and more than 7000 times speed increase using AFB compared to BF to schedule tasks with minimal hit to energy reduction and SLAV. We also empirically proved that the DCFB is getting a consistent speed increase compared to BF irrespective to the size of the problem. While ABF is getting larger relative speed increases for larger BPP problem. ABF also promises even larger speed improvement in the future with newer or Better GPU hardware. This breakthrough in speed increase in ABF compared to BF can finally bring online scheduling techniques algorithm into time feasible range for large sets due the very short execution time.

While it is hard to imagine any further speed increase that can be done to this algorithm there is still some room for improvement. Porting the algorithm to OpenCL allows the algorithm to run on a non-Nvidia GPU, thus bringing ABF into even more hardware. Adding an MPI option can make the algorithm better for Multi CPU solution or for Xeon Phi aided simulation. Adding more constraint, such as migration cost, RAM and Disk Utilization can make the model more accurate. Replacing the SLAV model with a queuing model is more realistic in real datacenters. Rewriting the algorithms to work on a global scale instead of local can get less speedups compared to ABF and DCBF, but it can slightly improve the energy saving and performance degradation.

## REFERENCES

- [1] X. W.-D. W. a. L. A. B. Fan, "Power provisioning for a warehouse-sized computer.," *ACM SIGARCH Computer Architecture News* 35.2, pp. 12-23, 2007.
- [2] C. Bash and G. Forman, "Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center.," HPL-2007-62 , August 23, 2007.
- [3] J. G. Koomey, "Estimating total power consumption by servers in the US and the world.," 2007.
- [4] Data Center Experts, "Data Center Cooling Efficiency & Containment," [Online]. Available: <http://www.datacenterexperts.com/products/data-center-cooling-efficiency-and-containment.html>. [Accessed 30 December 2014].
- [5] J. Koomey, "Growth in data center electricity use 2005 to 2010."Oakland, CA: Analytics Press.," August 1 2011.
- [6] HPL, "Tech Report: HPL-2003-5: Balance of Power: Dynamic Thermal Management for Internet Data Centers.," 2003-5.
- [7] M. el Mehdi Diouri, O. Gluck and L. Lefevre, "Towards a novel smart and energy-aware service-oriented manager for extreme-scale applications," *Green Computing Conference (IGCC), 2012 International* , pp. 4-8, June 2012.
- [8] M. Etinski, J. Corbalan, J. Labarta and M. Valero, "Linear programming based parallel job scheduling for power constrained systems," *High Performance Computing and Simulation (HPCS), 2011 International Conference on* , pp. 72-80, July 2011.
- [9] J. Aragon, J. Gonzalez and A. Gonzalez, "Power-aware control speculation through selective throttling," *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on* , pp. 103-112, 8-12 Feb. 2003.
- [10] A. a. R. B. Beloglazov, "Energy efficient allocation of virtual machines in cloud data centers.," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. IEEE*, 2010.
- [11] Y. Hotta, M. Sato, H. Kimura, S. Matsuoka, T. Boku and D. Takahashi, "Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster.," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 25-29, April 2006.
- [12] Y. Li, H. Zhang and Kyong Hoon Kim, "A Power-Aware Scheduling of MapReduce Applications in the Cloud," *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pp. 613-620, Dec 2011.

- [13] J. Guo and L. Bhuyan, "Load Balancing in a Cluster-Based Web Server for Multimedia Applications," in *Load Balancing in a Cluster-Based Web Server for Multimedia Applications*, Nov. 2006.
- [14] Q. Zhang, A. Riska, W. Sun, E. Smirni and G. Ciardo, "Workload-aware load balancing for clustered Web servers," *Parallel and Distributed Systems, IEEE Transactions on* , pp. 219-233, March 2005.
- [15] H. Topcuoglu, S. Hariri and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on* , , pp. 260-274, Mar 2002.
- [16] R. S. X. D. K. & T. J. Lewis, "An investigation into two bin packing problems with ordering and orientation implications," *European Journal of Operational Research*, pp. 52-65, 2011.
- [17] T. G. P. G. R. W. & T. R. Crainic, "Efficient lower bounds and heuristics for the variable cost and size bin packing problem.," *Computers & Operations Research*, pp. 1474-1482, 2011.
- [18] R. T. K. U. L. & W. U. Panigrahy, "Heuristics for vector bin packing," Microsoft Research, 2011.
- [19] Y. L. W. G. M. & d. S. R. Wu, "Three-dimensional bin packing problem with variable bin height," *European Journal of Operational Research*, pp. 347-355, 2010.
- [20] S. & L.-O. A. Kamali, "An All-Around Near-Optimal Solution for the Classic Bin Packing Problem," [Online]. Available: <http://arxiv.org/pdf/1404.4526v1.pdf>. [Accessed 2014 December 2014].
- [21] E. Duesterwald, J. Torrellas and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on* , vol., no., ], , " pp.220,231,27, Sept.-1 Oct. 2003.
- [22] D. Joseph, "Prefetching Using Markov Predictors," " , " *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on* , , pp. 252-263, 2-4 june 1997.
- [23] S. R. C. K. a. P. R. D. Economou, "Full-system power analysis and modeling for server environments," in *In Workshop on Modeling Benchmarking and Simulation* , June 2006.
- [24] R. N. e. a. Calheiros, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience 41.1* , pp. 23-50, 2011.
- [25] Amazon, Inc, "Amazon AWS," [Online]. Available: <http://aws.amazon.com/ec2/>. [Accessed 1 December 2014].
- [26] Apple. Inc, "iCloud," [Online]. Available: <https://www.icloud.com/> . [Accessed 1 December 2014].
- [27] Microsoft, Inc, "OneDrive," [Online]. Available: <https://onedrive.live.com/about/en-us/>. [Accessed 1 December 2014].



- [28] Google, Inc, "Google Drive," [Online]. Available: <https://www.google.com/drive/>. [Accessed 1 December 2014].
- [29] Dropbox, Inc, "Dropbox," [Online]. Available: <https://www.dropbox.com/>. [Accessed 1 December 2014].
- [30] Microsoft, Inc, "Microsoft Azure," [Online]. Available: <https://azure.microsoft.com/>. [Accessed 1 December 2014].
- [31] Openstack, "OpenStack Installation Guide for Ubuntu 12.04 (LTS) havana," OpenStack, 2013.
- [32] R. S. D. S. S. & T. J. Chheda, "Profiling Energy Usage for Efficient Consumption," *The Architecture Journal: Green Computing Issue*, 2008.
- [33] Cisco, Inc, "Server power calculator analysis: Cisco UCS power calculator and HP power advisor," Cisco Report, 2011.
- [34] R. Basmadjian, F. Niedermeier and H. De Meer, "Modelling and analysing the power consumption of idle servers," in *Sustainable Internet and ICT for Sustainability (SustainIT)*, 4-5 OCT,2012.
- [35] C. Dupont, G. Giuliani, F. Hermenier, T. Schulze and A. Somov, "An energy aware framework for virtual machine placement in cloud federated data centres," in ", " *Future Energy Systems: Where Energy, Computing and Communication Meet (e-Energy)*, 2012 Third International Conference on , 9-11 May 2012.
- [36] D. Leubke, "The Democratization of Parallel Computing," in *High Performance Computing with CUDA*, 2007.
- [37] Nvidia corporation, "NVIDIA's Next Generation CUDA compute architecture: Fermi," in *white paper*, 2009.
- [38] Nvidia corporation, "CUDA c Best Practice Guide," 1 August 2014. [Online]. Available: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3MwMrBaDa>. [Accessed 25 12 2014].
- [39] Techpowerup, "Nvidia GeForce Titan Z," [Online]. Available: <http://www.techpowerup.com/gpudb/2575/geforce-gtx-titan-z.html>. [Accessed 1 December 2014].
- [40] Techpowerup, "Nvidia GeForce GTX 760M," [Online]. Available: <http://www.techpowerup.com/gpudb/2305/geforce-gtx-760m.html> . [Accessed 1 December 2014].
- [41] Techpowerup, "Nvidia GeFore GT 740," [Online]. Available: <http://www.techpowerup.com/gpudb/1987/geforce-gt-740.html>. [Accessed 4 December 2014].