GPGPU Design Space Exploration Using Neural Networks

by

Ali Jooya
B.Sc., Azad University, 2003
M.Sc., Iran University of Science and Technology, 2009

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Ali Jooya, 2018
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

GPGPU Design Space Exploration Using Neural Networks

by

Ali Jooya
B.Sc., Azad University, 2003
M.Sc., Iran University of Science and Technology, 2009

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Amirali Baniasadi, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Mohsen Akbari, Outside Member
(Department of Mechanical Engineering)

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Amirali Baniasadi, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Mohsen Akbari, Outside Member
(Department of Mechanical Engineering)

ABSTRACT

General Purpose computing on Graphic Processing Unit (GPGPU) gained attention in 2006 with NVIDIA's first Tesla Graphic Processing Unit (GPU) which could perform high performance computing. Ever since, researchers have been working on software and hardware techniques to improve the efficiency of running general purpose applications on GPUs. The efficiency can be evaluated using metrics such as energy consumption and throughput and is defined based on the requirements of the system. I define it as obtaining high throughput by consuming minimum energy.

GPUs are equipped with a large number of processing units, a high memory bandwidth, and different types of on-chip memory and caches. To run efficiently, an application should maximize the utilization of GPU resources. Therefore, a good correspondence between the computing and memory resources of the GPU and those of application is critical. Since an application's requirements are fixed, the GPU's configuration should be tuned to these requirements. Having models to study and predict the power consumption and throughput of running a GPGPU application on a given GPU configuration can help achieve high efficiency.

The main purpose of this dissertation is to find a GPU configuration that best matches the requirements of a given application. I propose three models that predict a GPU configuration that runs an application with maximum throughput while consuming minimum energy.

The first model is a fast, low-cost and effective approach to optimize resource allocation in future GPUs. The model finds the optimal GPU configuration for different available chip real-estate budgets .

The second model considers the power consumption and throughput of a GPGPU application as functions of the GPU configuration parameters. The proposed model accurately predicts the power consumption and throughput of the modeled GPGPU application. I then propose to accelerate the process of building the model using optimization techniques and quantum annealing. I use the proposed model to explore the GPU configuration space of different applications. I apply multiobjective optimization technique to find the configurations that offer minimum power consumption and maximum throughput.

Finally, using clustering and classification techniques, I develop models to relate the power consumption and throughput of GPGPU applications to the code attributes. Both models could accurately predict the optimum configuration for any

given GPGPU application.

To build these models I have used different machine learning techniques and optimization methods such as Pareto Front and Knapsack optimization problem. I validated the model produced results with simulation results and showed that the models make accurate predictions.

These models could be used by GPGPU programmers to identify the architectural parameters that most affect an application's power consumption and throughput. This information could be translated into software optimization opportunities. Also, these models can be implemented as part of a compiler to help it to make the best optimization decisions. Moreover, GPU manufacturers could gain insight on architectural parameters which would profit GPGPU applications the most in terms of power and performance and hence invest on these.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **vi**

List of Tables **x**

List of Figures **xii**

List of Abbreviations **xiv**

Acknowledgements **xvii**

Dedication **xviii**

1 Literature Review **1**

1.1 Efficiency Metrics 3
1.2 GPU Architecture 3
1.2.1 NVIDIA GPUs 9
1.3 Performance: Challenges and Obstacles 14
1.3.1 Performance Modeling 19
1.4 GPU Design Space Exploration 23
1.5 Neural Networks 26

2 Introduction and Thesis Roadmap **31**

2.1 Benchmarks, Simulator and GPU Architecture 34
2.2 Design Space Exploration for Efficient Resource Allocation 35
2.3 Design Space Exploration using a NN-Based Power and Performance Predictor 36

2.4 Predicting Optimum Configurations Using Multi-Objective Optimization Method 37
2.5 Predicting Power-Performance Optimum Configuration Using Code Attributes 38
2.6 Obtaining a Good Configuration for a Set of Application Code 39
2.7 Outlier Detection Heuristics 39
2.8 Accelerating NN Ensemble Using Quantum Annealing 39

3 Efficient Resource Allocation in GPUs 41
3.1 Background 42
3.1.1 Plackett Burman Design Technique 42
3.1.2 Knapsack Problem 43
3.2 Methodology 44
3.2.1 Establishing the transistor count 45
3.2.2 Benchmarks 45
3.2.3 Use of Plackett-Burman and optimization 46
3.3 Evaluation and Results 47
3.3.1 Results 49
3.4 Related Work 51
3.5 Conclusion 52

4 Neural Network Based GPGPU Power and Performance Predictor 54
4.1 Background 57
4.1.1 GPU Architecture 57
4.1.2 Simulations and Benchmarks 57
4.1.3 Descriptor Vector Expansion 57
4.1.4 NN Ensemble 59
4.2 Methodology 62
4.2.1 Determining The Effective Design Space 62
4.2.2 Constructing the Model 63
4.2.3 Outlier Detection 64
4.3 Results 65
4.4 Conclusion 68

5 Accelerating Neural Network Training with Quantum Annealing 69
5.1 Optimization Methods 70

5.1.1 Exhaustive Search 70
5.1.2 Tabu Search 70
5.1.3 Quantum Annealing 71
5.1.4 SQA 72
5.2 Methodology 72
5.3 Experimental Setup and Results 73
5.3.1 Guided-selection Heuristic Alternative Results 74
5.3.2 Incorporating the Proposed Mechanism into the Ensemble Neural Network Model 76
5.3.3 Evaluating the Model with More Benchmarks 77
5.4 Conclusion 79

6 MultiObjective GPU Design Space Exploration Optimization 82
6.1 Pareto Optimal Optimization Technique 83
6.2 Methodology 84
6.2.1 Scaling the Power and Performance values 84
6.2.2 Quality of Obtained Predicted Pareto Front 85
6.2.3 Random Sets 87
6.3 Results 87
6.3.1 Obtaining a Good Configuration for a Set of Application Codes 89
6.4 Conclusion 91

7 Optimum Power-Performance GPU Configuration Prediction based on Code Attributes 93
7.1 Background 96
7.1.1 GPU Architecture 96
7.1.2 Simulations and Benchmarks 97
7.1.3 Profiling 97
7.1.4 Clustering 98
7.1.5 Classification 100
7.2 Methodology 100
7.2.1 Obtaining the Optimum Configurations from Predicted Super-Configuration 102
7.2.2 Evaluation 104
7.3 Experimental Setup and Results 105

7.3.1 K-Means Clustering 105
7.3.2 Classification 107
7.3.3 Analysis 107
7.4 Conclusion and Discussion 110

8 Conclusion and Future Work **111**
8.1 Future Direction 112

A Additional Information **114**
A.1 Integer Linear Programming 114
A.2 Outlier Detection Heuristic 116
A.3 Pareto Front Optimization 117
A.4 Kmeans Clustering 118

Bibliography **120**

List of Tables

Table 1.1 Main features of different generations of NVIDIA GPUs. 14

Table 2.1 GPGPU Benchmarks used to evaluate the proposed models. . . 35
Table 2.2 GPU configuration. 36

Table 3.1 Placket-Burman design with fold-over; 7 parameters, 16 experiments. 43
Table 3.2 Transistor cost (in millions of transistors) of resource units . . . 45
Table 3.3 Benchmarks and low and high values of resources 46
Table 3.4 GPU configuration. 47
Table 3.5 All configurations for region 2 of the NN benchmark 49
Table 3.6 Resulting parameter effects after Plackett-Burman for all benchmarks 50

Table 4.1 Range of variable GPU configuration parameters. 63
Table 4.2 The power and performance prediction results. 66
Table 4.3 The filter coverage and efficiency. 67
Table 4.4 The coverage and efficiency of the random filter. 67

Table 5.1 GPGPU benchmarks 74
Table 5.2 Distances and timings of different solvers 75
Table 5.3 NN-based predictor performance comparison for different methods 76
Table 5.4 Range of variable GPU configuration parameters. 78
Table 5.5 The power and performance prediction results. 80
Table 5.6 The filter coverage and efficiency. 81

Table 6.1 Pareto Front results summary. 89
Table 6.2 Overal optimum GPGPU configuration. 91

Table 7.1 Applications attributes obtained using SASSI. 99

Table 7.2 The number of configurations in each predicted Super-Configuration for different number of sub-ranges. 103
Table 7.3 Performance-optimum configurations and the corresponding sub-range numbers. 104
Table 7.4 Ranking score of the selected optimal configurations using K-means clustering method. 106
Table 7.5 Ranking score of the selected optimal configurations using classification method. 108

List of Figures

Figure 1.1 An Intel Core i7 processor architecture [40]. 5
Figure 1.2 Radeon HD 5870 GPU architecture [11]. 7
Figure 1.3 Execution model of a GPU kernel on NVIDIA (left) and AMD (right) GPUs. 9
Figure 1.4 NVIDIA's Fermi Architecture [92]. 11
Figure 1.5 a) SIMT model prior to Volta and b) Volta's SIMT model [15]. 13
Figure 1.6 NVIDIA GPUs road-map with the influential advances [38]. . . 15
Figure 1.7 A single-layer neural network structure [77]. 27
Figure 1.8 A multilayer neural network with one hidden layer with three neurons [77]. 28

Figure 2.1 Projection of the design space to (a) number of memory controller blocks and (b) constant cache size. 32

Figure 3.1 Performance of LIB benchmark for different sizes of DL1 cache and number of memory controller blocks 47
Figure 3.2 Optimum configuration regions for the NN benchmark. 49
Figure 3.3 Performance Comparison between the Model-suggested and the best performing configurations for different benchmarks 51

Figure 4.1 The block diagram of the NN ensemble. 60
Figure 4.2 (a) GME and (b) maximum error before and after applying the filter. 66
Figure 4.3 Filter efficiency. 68

Figure 5.1 Quantum annealing vs thermal annealing. 71

Figure 6.1 Actual and Predicted Pareto Front for STO benchmark. 88

Figure 7.1 Block diagram of the proposed model. 94

Figure 7.2 Mapping from configuration to Super-Configuration space. . . . 95
Figure 7.3 Obtaining the power-performance performance-optimum configuration from the predicted Super-Configuration. 95
Figure 7.4 Instrumentation flow of SASSI [85]. 98

Figure A.1 The optimum resource allocation for BFS benchmark. 115

List of Abbreviations

3D 3 Dimensional 1

APU Accelerated Processing Unit 6

BKP Bounded Knapsack Problem 44

CF Control Flow 98

CMOS complementary metal oxide semiconductor 18

CPU Central Processing Unit 1

CUDA Compute Unified Device Architecture 10

DL1 Data Level 1 47

DL2 Data Level 2 57

DRAM Dynamic Random Access Memory 16

DVFS Dynamic Voltage and Frequency Scaling 25

DWF Dynamic Warp Formation 17

EU Execution Unit 5

FP Floating Point 5

FPGA Field Programmable Gate Array 19

GD Gradient Descent 28

GME Geometric Mean of Error 77

GPGPU General Purpose computing on Graphic Processing Unit .... iv, 1

GPU Graphic Processing Unit .... iv, 1

HPC High Performance Computing 18

ID Identification 4

IL1 Instruction Level 1 57

ILP Instruction Level Parallelism 21

Int Integer 98

IPC Instruction Per Cycle 3

MCB Memory Controller Block 31

MIC Many Integrated Core 5

MIMD Multiple Instruction Multiple Data 3

MLP Memory Level Parallelism 21

MMKP Multi-dimensional Multiple-choice Knapsack Problem 52

MPSoC MultiProcessor System-on-Chip 52

NN Neural Network 26

nvcc NVidia Cuda Compiler 25

PB Plackett-Burman 35

PC Program Counter 8

PCIe Peripheral Component Interconnect Express 12

PDOM Post Dominant 17

PE Processing Element 4

PF Pareto Front 82

QMC Quantum Monte Calro . . . 72

QSAR Quantitative Structure–Activity Relationship . . . 55

RBF Radial Basis Function . . . 113

RF Register File . . . 101

SASSI NVIDIA assembly code SASS Instrumentor . . . 97

SC Streaming Core . . . 6

SDK Software Development Kit . . . 45

SE SIMD Engine . . . 6

SFU Special Function Unit . . . 6

SIMD Single Instruction Multiple Data . . . 3

SIMT Single Instruction Multiple Thread . . . 3

SM Streaming Multiprocessor . . . 10

SMO Sequential Minimal Optimization . . . 25

SQA Simulated Quantum Annealing . . . 72

SRAM Static Random Access Memory . . . 18

VLIW Very Long Instruction Word . . . 3

VPU Vector Processing Unit . . . 5

ACKNOWLEDGMENTS

There are many people to thank for their part in my success. First, I would like to thank my family for being there for me whenever I needed them. Their endless support and encouragement made all these years that I have been away from home bearable. I would like to thank my adviser, Dr. Nikitas Dimopoulos, for giving me a home in his lab and support over the years. I am grateful for his guidance and the opportunities he has afforded me. He has been life and PhD adviser to me. Also, I would like to thank my co-supervisor, Dr. Amirali Baniasadi, for giving me the opportunity to work with him and for his guidance through all these years. I would like to thank IT staff, lab technicians and graduate office staff for their continuous support. Last but not least, I would like to thank all my friends and many others for being so nice and supporting.

Ali Jooya

DEDICATION

I dedicate this work to my family for their constant support and encouragement during the challenges of graduate school and life. I am truly thankful for having them in my life.

Chapter 1

Literature Review

Graphic Processing Units (GPUs), as the name implies, were introduced to process 3 Dimensional (3D) graphics. Striving for higher performance, the power and thermal limitations of modern Central Processing Unit (CPU)s that prevented achieving higher clock speed, and the computational capability of GPUs motivated the idea of performing GPGPU.

NVIDIA, AMD and Intel are three major manufacturers of GPUs. The number of top 500 supercomputers that were equipped with NVIDIA GPUs increased from 3 to 100 from Jun 2010 to June 2018 [87].

Transforming a CPU code into a GPU kernel[1], often requires familiarity with GPU architecture, major code rewriting and optimization to obtain good performance. As GPUs were designed to process graphics applications, employing them for general purpose computing may not necessarily result in achieving speedup.

The challenge of GPGPU programming comes from the fact that GPUs were designed to process streams of independent vertices and texels[2] utilizing thousands of simple processing elements. Applications need to meet two key attributes of computer graphic computation to map well on GPUs and obtain good performance; data parallelism and independence [32]. Applications that meet these requirements may still suffer from data flow and memory access divergences as performance barriers. Also, performance of GPGPU applications suffer from control flow divergence [50]. In GPGPU applications, threads running in lock-steps may diverge at control flow instructions which results in serial execution of two groups of the threads taking dif-

[1] A kernel is a function that runs in parallel on a GPU.

[2] texture elements. Texture is the smallest graphic elements which is the digital representation of the surface of an object.

ferent execution paths. This results in under-utilization of computational resources and poor performance. Many hardware and software techniques have been proposed to address these performance barriers and increase the efficiency by exploiting application parallelism and efficient thread and memory access scheduling.

To indicate how efficient the execution of an application on a GPU is, one can look at the performance per watt which is the amount of computation the GPU delivers by consuming a watt of power. GPUs can reach high performance per watt [52] if the application's parallelism and input data size are large enough to utilize the computing resources and memory bandwidth of the GPU. The achievable performance per watt depends on the applications characteristics, i.e. computational and memory demands, and the compute and memory resources of the GPU. If there is a balance between the two, the application runs efficiently on the GPU.

Our work focuses on exploring application-specific GPU architectures that best meet compute and memory requirements of the application. To achieve this, we explore GPGPU design space, model power and performance of GPGPU applications running on different GPU configurations, develop models to predict application-specific power-performance optimum GPU configurations. We propose a method to obtain a configuration that a set of applications benefit from it. We propose techniques to speedup the execution time of the model. Also, we develop a model that predicts application-specific performance-optimum configuration based on the attributes of the application code.

Owens et al. in 2008, presented a remarkable paper on GPU computing, its architecture, programming model, tools, programming languages and applications [71]. In this chapter we build-up on Owens work and present a review of recent research on GPGPU.

This chapter addresses the current and future trends and challenges in modeling, measuring and improving performance and energy efficiency and present innovations in GPU architecture that broadened GPGPUs applicability by making it possible to run irregular applications.

The rest of this chapter is organized as follows. In Section 1.2 we review GPU architectures. In Section 1.3 we review literature on GPGPU applications performance measurement, analytical performance modeling, and performance improvement techniques. In Section 1.4 we review previous works on GPGPU design space exploration methods. Section 1.5 reviews the basics of artificial neural networks.

1.1 Efficiency Metrics

In different models that we present in this dissertation we evaluate the efficiency that an application can achieve running on different GPU architectures. Therefore, it is important to define these metrics. We consider two metrics, the average power consumption and performance for which we measure Instruction Per Cycle (IPC).

Power is the rate of energy, i.e. the energy consumed per unit of time (Joules per second) and its unit is Watt. For each application running on a GPU architecture, we report the average power that is consumed for the entire execution of the application. We use the term "*power*" instead of "*the average power consumption*" in the rest of this dissertation. The average power we report includes both dynamic and static power consumption.

To measure the performance, we use IPC which depends on the benchmark and the architecture, but not on the technology and represents the rate of computation. It is worth mentioning that we obtain the power consumption for a particular technology and we keep the technology constant while we explore the architecture. The results may be altered if the technology changes. However, the methods will still be applicable. We use the terms "*performance*" and "*IPC*" interchangeably in the rest of this dissertation.

1.2 GPU Architecture

GPUs exemplify the SIMD paradigm; Single Instruction Multiple Data (SIMD) architecture that exploits data-level parallelism by executing a single instruction on multiple processing units.

The main difference between the major vendors' GPUs or accelerators is in the parallel execution engine design. All the three vendors' (NVIDIA, Intel and AMD) devices employ multiple SIMD units that implement hybrid Multiple Instruction Multiple Data (MIMD)/SIMD structures. Intel uses vector processing units as the core of its accelerators compute engine while NVIDIA and AMD use Single Instruction Multiple Thread (SIMT) and Very Long Instruction Word (VLIW) semantics, respectively.

A Vector processor exploits data level parallelism by applying the same operation on a set of independent data items in parallel. Vector processors operate on long data vectors with a single instruction to reduce the required fetch and decode bandwidth.

For example, vector add instruction adds two long vectors and stores the results in third vector. Vector processors were first used in supercomputers to process applications with large vector size. As the functional unit width of the processors were smaller than the data vector length, it took multiple clock cycles to complete a single vector instruction. Therefore, to increase the performance, the functional units were deeply pipelined. Scientific and graphic applications are good fit to vector processors as they have large amount of data level parallelism which can be presented in form of vectors and matrices [39].

The SIMD execution model is very similar to vector processing as in both a single instruction is performed on multiple independent data elements. However, in SIMD, wider functional units, e.g. 256-bits or 512-bits wide, are used which can process vector instructions in one cycle. Therefore, the functional units are not deeply pipelined that makes them more efficient to operate on shorter vectors compared to vector processors [39].

The Single Instruction Multi-Thread (SIMT) execution model, introduced by NVIDIA in 2006, is similar to SIMD as both architectures implement parallelism by dispatching the same instruction to multiple execution units. However, in SIMD all the elements of the vector execute together, whereas in SIMT multiple threads start the execution together and it is possible for individual threads to have their own execution path. SIMT architecture exploit both inter-task level parallelism as well as data level parallelism [53].

The difference between Vector processors and SIMT architecture stems from the way they manage parallel data. In SIMT architecture data level parallelism is converted into thread level parallelism and the programmer only describes the behavior of a single thread and creates as many copies of that thread as needed. Each thread has a unique Identification (ID) which is used to access the data and can have its own program flow.

However, in Vector processors a single thread executes each instruction for all data elements of vectors or matrices on a vector of Processing Element (PE)s. In vector programming, the programmer schedules operations on PEs and there is no need to use synchronization instructions. Each instruction is executed on all PEs in lock steps. The disadvantage of this is that the programmer needs to know the vector lane[3]. size. However, in SIMT programming model, SIMT lane size is transparent to the programmer and the hardware scheduler decides on the execution order and schedules

[3] A vector lane is a functional unit with registers

operations on SIMT lanes. Therefore, the programmer needs to use synchronization instructions wherever the threads need to be synchronized [36].

Memory system efficiency is another difference between the two architectures. In vector processors, vector strides are important. Vector processors rely on memory interleaving to supply the huge data bandwidth. If striding is not designed carefully, then inefficiencies occur. In SIMT architectures, irregular memory accesses causes the threads to stall on cache misses. Extensive hardware multi-threading mechanism is used to hide memory system latency.

Intel's graphic processing units [40] and Many Integrated Core (MIC) architecture [20] support executing compute kernels. The graphic processing units are integrated into the same die that the processors reside while the MIC architectures are available as separate chips.

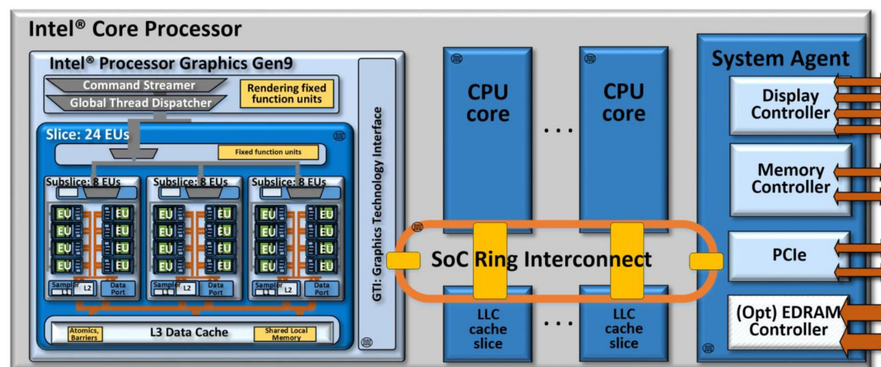


Figure 1.1: An Intel Core i7 processor architecture [40].

Figure 1.1 shows Intel Core i7 processor die that includes four CPU cores, a graphic processing unit (aka Processor Graphic), a shared last level cache, memory and display controllers [40]. Each Processor Graphic includes one or more slices each of which pack 3 subslices. Each subslice has 8 Execution Unit (EU)s that perform SIMD computation. EUs combine Simultaneous Multi-Threading (SIMT) and SIMD execution to exploit the parallelism in graphic applications and compute kernels. On every cycle, each EU can execute up to 16 32-bit Floating Point (FP) operations from up to four different threads.

Intel's MIC [4] architecture uses multiple in-order x86 cores each of which implements a wide vector processor and scalar unit. Intel's Vector Processing Unit (VPU)s execute integer, single and double precision floating point instructions. Consistent

[4] Many Integrated Core

memory hierarchy also increases the efficiency of inter-processor communication, data sharing and synchronization. Employing x86 CPU cores, Intel accelerators support page-faults, irregular application execution and ease the programmability. As CPU cores support x86 instruction set, the accelerator can run existing codes, operating systems and applications.

AMD's Accelerated Processing Unit (APU) combines one or more x86-based CPU core(s) with a GPU architecture in a single package to increase performance and energy efficiency [11]. AMD APUs run parallel applications on hundreds of cores distributed among SIMD Engine (SE)s. Each core executes a thread and implements a five-way VLIW processor. It means five independent instructions from a thread can be executed in each clock cycle if the required execution units are available. Therefore, they are able to exploit both instruction and thread level parallelism. The performance of VLIW processors can be limited by the amount of instruction level parallelism of the application.

As we mentioned before, the main architectural difference between NVIDIA and AMD GPUs is that AMD uses Very Long Instruction Word (VLIW) processing elements [101]. The important performance parameter in VLIW architecture is the packing ratio which is the ratio of m over n where m is the number of instructions the compiler can fill in a n-way VLIW processor. A packing ratio equal to one brings the highest performance. The other difference is in the memory system configuration. For example, the Radeon HD 5870 has level-1 (L1) and level-2 (L2) on-chip caches which are used to capture constants and image objects. Fermi does not carry the limitation on the type of data its L1 and L2 caches can store.

Figure 1.2, Shows the architecture of the Radeon HD 5870. It is composed of 20 SIMD Engines (SEs) each of which can process 248 concurrent set of thread groups from multiple kernels. Each SE is equipped with 16 Streaming Core (SC)s, a branch control unit, 32KB local data cache and has access to a 8KB read-only L1 cache.

Each SC consists of four ALUs, one Special Function Unit (SFU), an operand preparation logic and a set of general purpose registers. The thread dispatch processor schedules a 64-thread warp to a SE. As each SE has 16 SC, it takes four clock cycles to execute an instruction from all the threads in the warp. Each SC executes a five-way VLIW instruction from a thread.

Like AMD, NVIDIA GPUs powered by SIMT engines. In NVIDIA GPUs SIMT engines are implemented by packing multiple homogeneous Processing Elements (PEs)

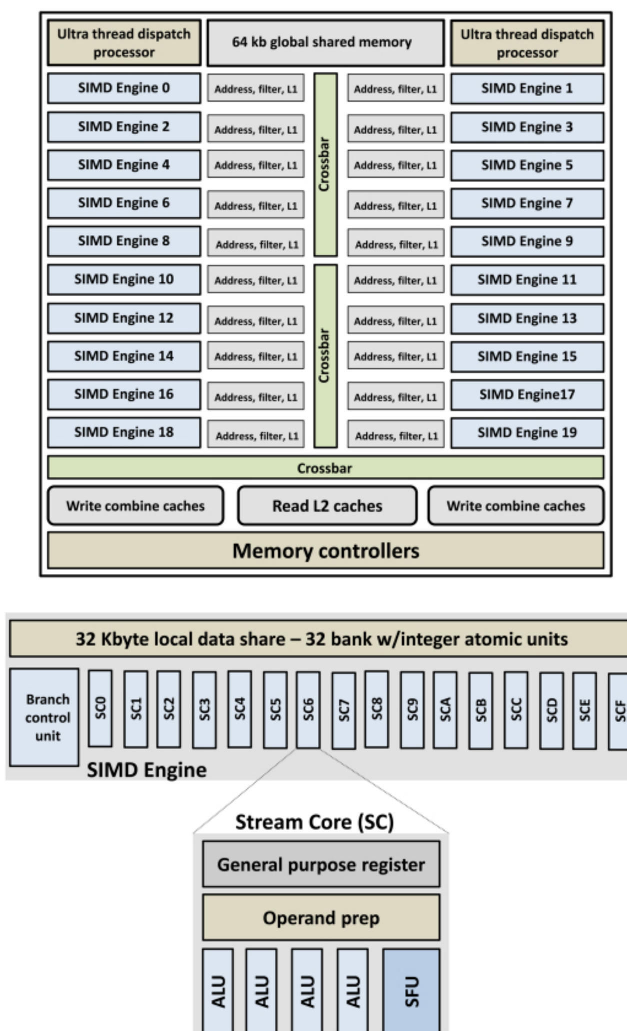


Figure 1.2: Radeon HD 5870 GPU architecture [11].

in a unit called Streaming Multiprocessor[5] (SM or SMX). Figure 1.4 shows NVIDIA's Fermi GPU architecture. The number of PEs in each SM varies for different NVIDIA GPUs but the SIMT width is 32 for current GPUs. Therefore, each SM may include more than one SIMT engines. The PEs of a SM can communicate through a memory subsystem which is visible to all PEs of the multiprocessor. PEs in different SMs communicate via a memory subsystem that is shared by all multiprocessors. A DRAM memory holds the GPU kernel and is shared with all multiprocessors. We will present a brief history of NVIDIA GPUs later in this section.

[5] As Streaming Multiprocessors execute shader programs such as pixel or vertex shaders, NVIDIA uses the term shader core to refer to a streaming multiprocessor. We use these two terms interchangeably.

GPGPU applications usually start their execution by running a single thread on CPU which is more power and performance efficient in executing applications with low amount of parallelism. When the application's execution reaches a parallel section, which is called a GPU kernel, CPU launches the kernel execution on the GPU to take advantage of its enormous computing power. When the kernel execution is finished and the results are available to the CPU, the rest of the application is executed on the CPU until it reaches another kernel or the execution is finished.

A kernel is the set of instructions inside a loop and each iteration of the loop is converted into a thread (NVIDIA) or work-item (AMD). Each thread has a unique ID which is used to access the data stream associated with the thread. Threads of a kernel are divided into groups called thread-blocks (NVIDIA) or work-groups (AMD). The size of thread-blocks vary in different architectures. For example, NVIDIA's Fermi supports thread-block size of maximum 1536 which is divided into 48 32-thread warps. Thread-blocks are further organized into smaller groups called warps (NVIDIA) or wavefronts (AMD). The thread scheduler broadcasts threads to multiple execution units of the SIMT pipeline. Figure 1.3 shows how threads of a GPU kernel managed and organized and indicates the terms NVIDIA and AMD use for different thread hierarchies. As NVIDIA's GPUs are widely used in supercomputers, desktops as well as academic studies, from now on, we shall use the terminology NVIDIA uses.

Each warp has one active Program Counter (PC)[6] and threads in a warp are executed on the PEs of the SM in lock step that means the execution of the threads proceeds together. In a given clock cycle, the active PC is used to fetch and execute an instruction for all the threads in the warp. The PE assigned to each thread accesses a portion of the register file associated to its thread to obtain the required data. All the threads in a warp execute the same instruction in a given clock cycle until they reach a control flow instruction.

When threads execute a control flow instruction, such as a conditional branch, some of the threads may evaluate the condition false while the reset evaluate it true. So, at this point, which is called divergence point, the execution of the threads take different execution path and, therefore, their execution is serialized. At the point of divergence, the threads that evaluated the condition false, execute a set of instructions until their execution path merges back with the rest of the threads in the warp at re-convergence point. At this point they reconverge with the rest of the treads of the warp and continue the execution in lock-step again. Diverging branches degrade the

[6]The latest NVIDIA architecture , i.e. Volta, maintains execution states per thread.

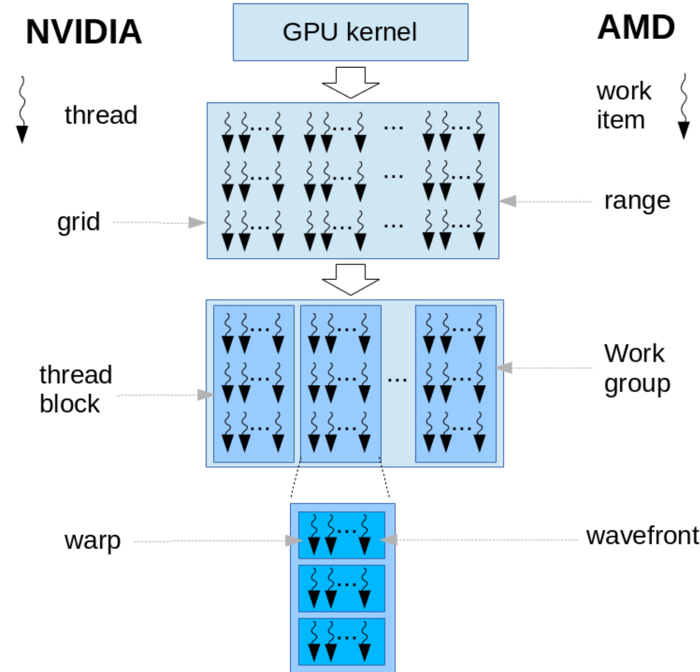


Figure 1.3: Execution model of a GPU kernel on NVIDIA (left) and AMD (right) GPUs.

performance due to under-utilization of processing elements.

The other performance obstacle in GPGPU is the memory system latency. GPUs employ hundreds of threads in each SM to hide memory system latency (DRAM memory access takes up to 400-600 cycles). When the execution of a warp stalls on a cache miss, while the memory system retrieves data from DRAM memory, the warp scheduler of the SM swaps the waiting warp with a ready warp. If an application has irregular memory access pattern or does not have enough computation to hide the memory system latency with, the achievable performance will be limited due to memory system response time.

As NVIDIA GPUs are widely used in desktop computers as well as servers and high performance systems, and because of the publicly available performance simulators, most of the researchers use NVIDIA GPUs as their computational platform. Therefore, we will present more details on NVIDIA GPU architectures.

1.2.1 NVIDIA GPUs

GPUs introduced before 2006 were dedicated to process graphics applications. GPU pipelines were composed of stages (i.e. vertex shader, rasterizer, pixel shader and

texture unit) specialized in processing certain types of operations. 3D objects are composed of triangles. To draw a triangle, a vertex shader receives the position and color of vertices and draws the triangle on the screen. A rasterizer receives the output of the vertex shader and determines which pixels on the screen are covered by the triangle and interpolates the geometric values for all the pixels covered by the triangle. A pixel shader determines the color and transparency of each pixel. And the texture unit maps the texture to the surface of an object.

As the technology size shrunk and more transistors could be packed into a single die, with every new generation of its GPUs, NVIDIA included improved arithmetic, more PEs, large caches, higher memory bandwidth, support for more concurrent threads, higher clock frequency etc. Beside these, some technological innovations and advances were introduced with some of the GPUs which we review in this section.

In 2006, NVIDIA took the first step toward high performance parallel computing by unifying the vertex and pixel shaders [54]. In the new architecture, called Tesla, the graphic pipeline is physically a recirculating path that visits the processors multiple times with some fixed-function tasks, e.g. rasterizer, in between. This enabled the hardware to process non-graphic applications. GeForce 8800, that was built based on Tesla architecture, executed up to 768 concurrent threads (or up to 24 warps) per Streaming Multiprocessor (SM) using hardware multi-threading with zero scheduling overhead. To improve the utilization, the Gigathread Engine was introduced with GeForce 8800 which fetches data from system memory to GPU memory, creates and distributes thread blocks to SMs.

With GeForce 8800, NVIDIA introduced a parallel computing platform called Compute Unified Device Architecture (CUDA). CUDA is an extension to the C programming language and enables software programs to perform operations on both CPU and GPU. Using CUDA platform, the programmer determines the functions to be run on the GPU (GPU kernel) and the size of the kernel. The size of the kernel is determined by the number of thread-blocks and the number of threads per thread-block of the GPU kernel. Thread-blocks are assigned to SMs and will run in parallel. Inside a SM, the threads of the thread-block are divided into groups of 32 threads (warps) and the warp scheduler of each SM, determines which warp runs at any given cycle. Based on the compute capabilities of the target GPU, one or more thread-block can be assigned to each SM and one or more warp can be run side by side. When the execution of a warp stalls, e.g. due to data missed in a local cache, the warp scheduler schedules another warp. This way, the memory latency is hidden

by computation that increases the performance [80].

The GTX 200 GPU families maintained the basic architecture of Tesla. A double-precision 64-bit floating point unit added to each SM which favored general computing on GPUs.

NVIDIA introduced a new GPU architecture code-named Fermi, with new features for both graphic and general purpose computing in 2009 [12, 92]. The major architectural improvements of the Fermi includes full memory system hierarchy, unified address space (64-bit virtual address and 40-bit physical address), two level thread scheduling, Error Correction Code (ECC) protection and concurrent execution of different kernels of the same application. Figure 1.4 shows the Fermi architecture (we don't display graphic computing components in the figure). Fermi's memory system support paged memory with multiple page sizes which improve heterogeneous computing by sharing data with system memory.

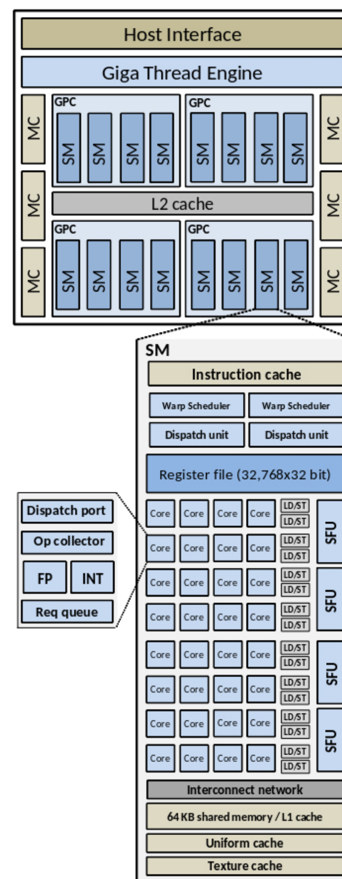


Figure 1.4: NVIDIA's Fermi Architecture [92].

NVIDIA introduced dynamic parallelism, Hyper-Q, grid management unit and

NVIDIA GPUDirect with Kepler architecture in 2012 [67].

With dynamic parallelism any kernel can launch another kernel and can manage the stream creation, events and dependencies without CPU intervention. This reduces the kernel launch overhead and eliminates the communication between the CPU and GPU for kernel launch.

Fermi supports up to 16 concurrent kernel launches from different streams[7]. The Hyper-Q feature of Kepler allows up to 32 work queues to be established between the GPU and the host CPU. This feature enables multiple kernels to be executed concurrently maximizing the utilization and the performance.

GPUDirect enables GPUs in the same computer system or GPUs across a network to access the GPU memory directly without the host CPU or memory intervention. When two GPUs on the same node want to communicate, the data should be transferred first to CPU memory and then to the destination GPU's memory. With GPUDirect the data will be transferred from the source GPU's memory directly to destination GPU's memory. In case of data transfer between GPUs on different nodes, the performance will be improved by eliminating the unnecessary copy of the data in host memory.

With Tesla P100, NVIDIA introduced the Pascal architecture in 2016 [14]. Pascal is equipped with NVLink, on-chip stacked memory and a Unified memory technologies.

NVLink is NVIDIA's high speed interface for GPU-to-GPU communication that provides 160 Gigabytes/second, 5x higher bandwidth compared to third generation of Peripheral Component Interconnect Express (PCIe) communication bus. The on-chip stacked memory improves efficiency and computational throughput by providing higher capacity and bandwidth. Programming Tesla P100 is simplified by unifying virtual address space for CPU and GPU memory.

Supporting half-precision floating point operation makes Pascal ideal for Deep Neural Networks in which the accuracy of double or single precision is not needed as additional noise in neural networks has been shown to improve accuracy [3]. This reduces memory usage of the neural network which makes training of larger neural networks possible. Also, it doubles the performance, compared to single precision performance which boosts the training of the neural networks.

The most recent architecture introduced by NVIDIA is Volta [15]. Tesla V100

[7]In CUDA, a sequence of operations that execute in the order they are issued in the source code. Operations in different streams can be interleaved or run concurrently.

is a GPU built on Volta architecture and featured new Tensor cores and new SIMT model. With the 640 Tensor cores packed in each GPU, the V100 boosts deep learning applications by performing 64 floating point fused multiply-add operations per clock per core. Tensor cores are custom-designed to perform matrix-matrix multiplication which is the core of neural network computation. Each Tensor core performs a 4x4 matrix multiplication operation per cycle ($D = AxB + C$).

In SIMT execution model, as we mentioned earlier, each warp (a group of 32 threads) shares one PC and when the executions of threads diverge at a control flow instruction, the executions of the threads branch into taken and not-taken paths and run serially until they reach to the reconvergence point where all the threads merge together again. The serialization of the execution of branching threads degrade the performance and concurrency. The new SIMT model maintains execution states, i.e. PC and call stack, per thread. The executions of the threads taking different paths at control flow instructions interleaved as shown in Figure1.5. Programmer may enforce reconvergence after instruction "Z" by explicitly calling sync function in the code.

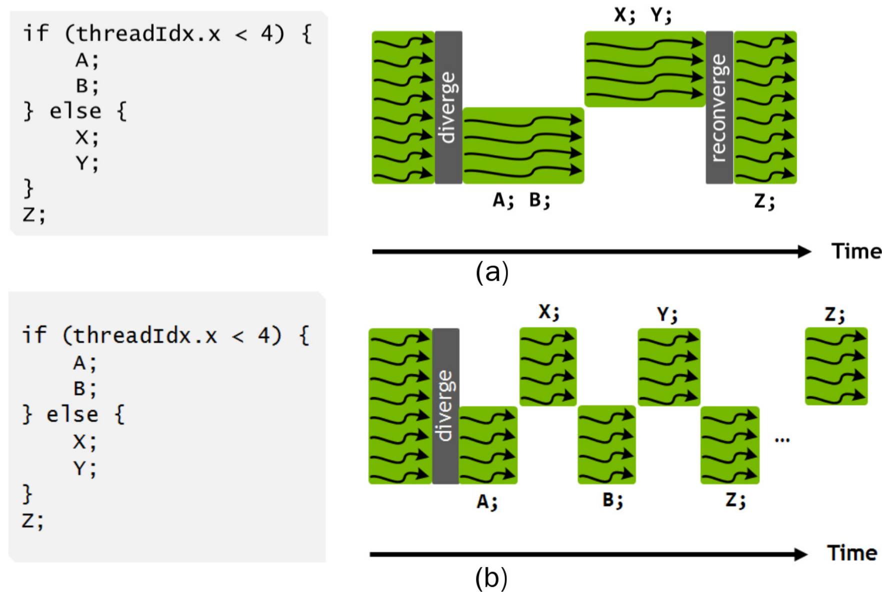


Figure 1.5: a) SIMT model prior to Volta and b) Volta's SIMT model [15].

Table 1.1 highlights the main features of each generation of NVIDIA GPUs. The main trend that NVIDIA GPUs have been following through the past decade can be summarized as follows. A) Reducing memory latency, by moving the DRAM closer to GPU cores, increasing memory bandwidth and introducing larger caches, B) increasing the number of processing elements, C) increasing GPU utilization by

facilitating parallel kernel execution and increasing the number of SMs, D) improving GPGPU programming user experience.

Table 1.1: Main features of different generations of NVIDIA GPUs.

	GeForce8800	GTX200	Fermi
Computation	Unified pipeline 8 SMs, 8 cores each 768 threads per SM	16 SMs, 16 Cores each 1024 threads per SM DP 64-bit FP units	15 SMs, 32 Cores each 1536 threads per SM 2 warp scheduler per SM
Memory	Off-chip DRAM	Off-chip DRAM	Off-Chip DRAM L1 and L2 caches Unified address space Paged memory
CPU-GPU Connection	PCIe	PCIe	PCIe
Parallelism			Up to 16 concurrent kernels
Programming	Introducing CUDA		

	Kepler	Pascal	Volta
Computation	16 SMs, 192 cores each 2048 threads per SM 4 warp scheduler per SM	56 SMs, 64 cores each 2048 threads per SM 2 partitions per SM	84 SMs, 168 cores each 4 warp scheduler per SM 4 partition per SM Tensor Cores for deep learning
Memory	GPUDirect	on-chip stacked memory	L0 cache on-chip stacked memory
CPU-GPU Connection	PCIe	NVLINK	NVLINK
Parallelism	Dynamic parallelism Hyper-Q	New SIMT model	Independent thread scheduling
Programming		Unified virtual address space	

Figure 1.6 shows the NVIDIA GPUs road-map including the main advances and the performance per watt that they offer [38].

1.3 Performance: Challenges and Obstacles

Employing hundreds of processing elements, the peak performance of GPUs is higher than that of conventional CPUs with relatively the same power demand. However, while executing general purpose applications, achievable performance on a GPU does

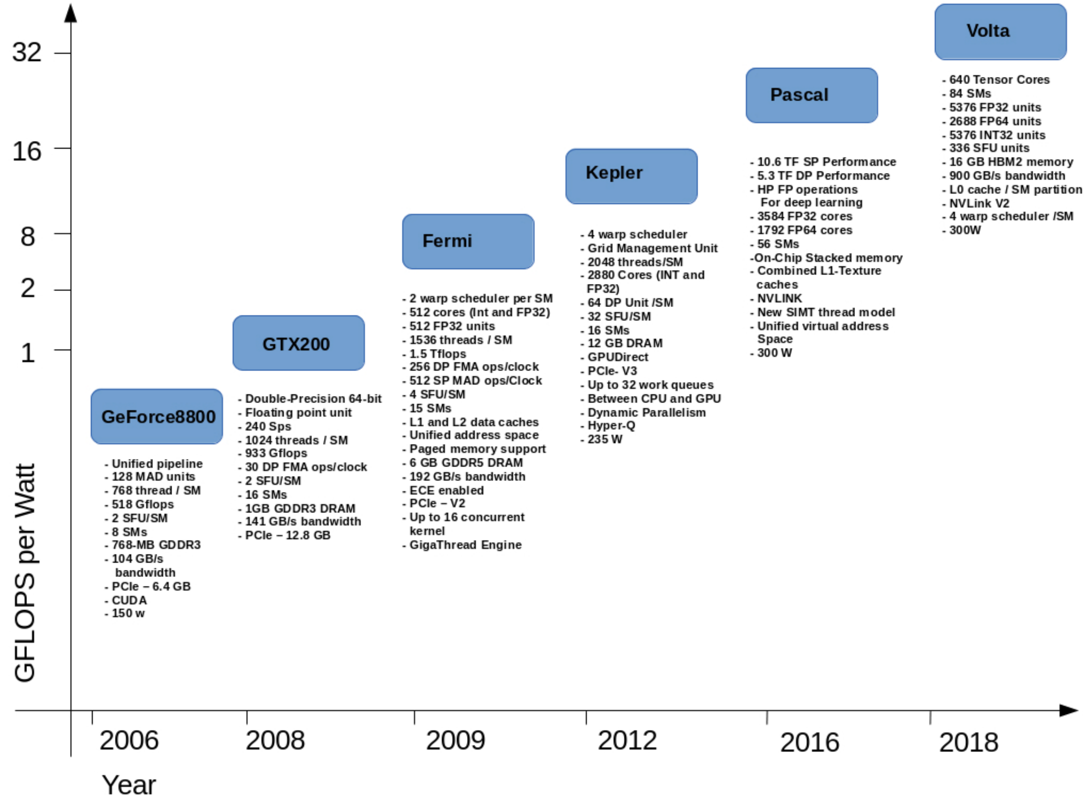


Figure 1.6: NVIDIA GPUs road-map with the influential advances [38].

not always surpass the performance of its CPU counterpart [61]. For example, if there is not enough parallelism in the application or the memory access pattern of the application is irregular the performance will be degraded.

The current GPUs are used as co-processors in CPU-based systems. A GPGPU application code is composed of serial code sections and parallel code sections. The execution of the application always starts with a serial section that runs on the CPU. When the execution reaches the parallel section, i.e. the GPGPU kernel[8] the CPU launches the kernel execution on the GPU. When the GPU is running the parallel section, the CPU can execute other tasks or other sections of the application that do not depend on the results from the GPU. When the kernel execution is finished and results become available to the CPU, the CPU continues the execution of the remaining of the application. GPGPU applications may have multiple serial sections and GPU kernels [80].

GPU and CPU memory are usually separate and data transfer happens through

[8]In GPGPU terminology the parallel section of code launched on GPU is called *Kernel*.

either PCIe port or NVLINK interconnection system[9] [14]. This introduces communication overhead that usually cannot be hidden by computation. Therefore, to gain speedup, the amount of computation that an application performs should be higher than the communication overhead.

The memory system in GPUs is composed of different components, e.g. level one and level two data caches, shared and texture memories. But, unlike CPU memory that is designed for short response time, GPU memory is designed for maximum bandwidth. The reason is the number of concurrent threads of GPU kernels that are much higher than that of CPU applications. This enables GPUs to hide memory latency with computation and shift the design goal to higher bandwidth rather than shorter response time. To compensate for the large global memory response time, the GPU coalesces global memory leads and stores issued by threads of a warp into as few memory transactions as possible [33]. Memory access requests of a warp to Dynamic Random Access Memory (DRAM) locations issue a single memory request if the memory locations that are accessed are consecutive and aligned. Kernels that have irregular memory access cannot take advantage of memory coalescing and as such their performance suffer from high memory latency because of the serialization of the memory requests, specially if the kernel is memory intensive.

As we mentioned earlier, the execution path of threads in a warp may diverge at control flow instructions. The execution of taken and not-taken paths serialized until all the threads in the warp reach to the reconvergence point. Therefore, the amount of speedup an application can gain by running on a GPU is limited by the amount of the application's parallelism, the memory access patterns and the control flow irregularities. Later in this section we review literature that addresses these issues.

How much performance should be expected by adding a GPU to a system is usually hard to predict before implementing the GPU version of an application and running it on an actual GPU. Vuduc et al. [89] studied the limitations of GPU accelerators (NVIDIA's Tesla) and showed that the achievable performance gain by adding a GPU to a system may not be as high as expected. To this end, the tuned multithreaded and the GPU version of three scientific applications were developed and their performances were compared. The results indicated that a well-tuned multithreaded version of an application running on a multicore CPU (Intel's Nehalem) may bring performance

[9]NVLINK introduced in 2016 with Tesla P100 for high-speed GPU to GPU and GPU to CPU communication [15].

close to that of a system with a GPU with roughly comparable energy consumption.

The limitations inherent in GPUs' architecture, i.e. the need to transfer data between system memory and GPU memory through PCIe[10] and the characteristics of applications, e.g. the amount of the available computation, memory access patterns and the complexity of the data structures, limits the performance that a GPU can deliver. However, the limited number of evaluated applications makes any general conclusion inappropriate. Moreover, this conclusion may not remain valid for future architectures. For example, fusion architectures eliminate the communication over PCIe as well as the sharing of the last level of memory system provides order of magnitude higher bandwidth for data transfers between CPU cores and the GPU. New GPU architectures like Volta communicate with CPU through high bandwidth NVLINK interconnection instead of PCIe. And the GPU memory can be implemented on the GPU die to further reduce the communication between CPU and GPU. Beside, multicore processors of the recent years include many more cores and run on higher frequency compared to Nehalem processor. Therefore, the conclusion made by this study may not hold true for the new computing platforms.

As we have studied so far, not all applications can benefit from GPUs compute power. An insight on the performance obstacles that prevent applications reaching the peak performance of the GPU helps the designers and the developers optimize the architecture and the applications for higher performance. Lashgar et al. studied three performance obstacles in GPUs, i.e. branch divergence, memory access latency and limited workload parallelism [50].

Various machine models have been made to predict potential obtainable performance gain by eliminating each of these performance barriers. Considering different control flow handling mechanisms, i.e. Post Dominant (PDOM), Dynamic Warp Formation (DWF)[11] ideal control flow, limited/unlimited hardware resources and ideal/non-ideal memory system, 12 machine models were evaluated. The results indicated that memory is the most influential performance parameter.

The performance gap between different machine models was studied to indicate how each parameter could potentially impact performance. For example, for each control flow mechanism, three performance gaps exist. First, the performance that can be achieved under an ideal memory system (memory potential). Second, the

[10]Before the introduction of NVLINK, discreet GPUs used PCIe buses to communicate with host CPUs.

[11]Post-dominator and Dynamic warp formation are two mechanisms introduced to handle diverging control flow instructions [27]

performance that can be achieved when the branch divergence is eliminated (control potential). Third, the performance achievable when the SM resources are unlimited (resource potential) [50].

The study showed that the performance gain from solutions that address memory inefficiency outweighs the performance gain obtained when investing in SMs resources or utilizing more advanced control flow mechanism. The results shows that, as of 2011, GPUs lose 40%, 15% and 2% of their potential performance when they employ a realistic memory system, control flow mechanism and limited SM resources, respectively [50]. This study was done 7 years ago. To see if the argument is still valid, we need to check how the ratio between performance and memory bandwidth of GPUs has changed over the past seven years. Since the time this study was done, GPUs memory bandwidth has increased around 2.5 times from the Kepler to the Volta and, due to smaller transistor size, the number of processing elements increased around 4.5 times [13, 15]. Therefore, we expect that the conclusion made by this study hold valid for newer generation of GPUs.

For the past ten years, GPUs have dominated the High Performance Computing (HPC) era and researchers proposed hardware and software solutions to overcome performance obstacles. As the technology size is reaching its limitation [90], it becomes important to see which path HPC will take from here. In [90], William argues that reaching to the end of Moore's law [62] brings up more innovations based on recently emerging technologies. The author predicts that 5-nm is where the technology size stops scaling down. Up until now, the industry and intellectuals putting all their efforts to scale silicon complementary metal oxide semiconductor (CMOS) and when this process stops, every advance will require significant discoveries in physics, biology and computer science. Techniques such as 3D integration or stacking might be a short solutions, but it seems unlikely for the manufacturers to be able to continually double the number of layers in a stack because of power and temperature issues.

One of the possible solutions by the author is memory-driven computing. Instead of using the traditional memory and storage hierarchy, i.e. high capacity slow flash drives for mass storage, medium range capacity and speed DRAMs for main memory and high speed low capacity Static Random Access Memory (SRAM) for caches, we should use a nonvolatile, high density, low power, low cost memory. With the memory structure of today's computers, 90% of a computer's work is to shuffle data between different memory components instead of doing computation. Replacing this hierarchy with a one level only nonvolatile memory to hold the entire dataset for a system can

eliminate this wast of time and energy fort data shuffling.

The other solution is to improve the interconnection performance by using photonic interconnect. The third solution offered by the author is to build specialty accelerators to address classes of computation and computers that are customized for particular problems. Even significant blocks of code can be translates into ASIC [12] hardware. Using these techniques one will be able to get continuous performance improvement by increased customization that uses photonic interconnect and memory driven computing.

In [7], Blott studied the role that Field Programmable Gate Array (FPGA)'s will play in future for high performance computing. As the current architectures are suffering from high power density and cooling problem, FPGAs can achieve higher energy efficiency through customized datapaths and memory architectures. With the new advances in software centric design environment for FPGAs e.g. Xilinx's OpenCl based design tool [94], will make FPGA design more accessible to HPC community. Both William's and Blott's research [7,90] suggest that the future of high performance computing relies on highly customized hardware.

Further, the first implementation of Memristors (Memory Resistor) in 2008, has led some researches to believe that they (memristors) will bring a new wave of innovations, specially in packing more transistors in smaller packages [9].

1.3.1 Performance Modeling

Given a GPU architecture, there is a considerable performance difference between a well-optimized kernel code, in which the thread block size and shared memory access pattern is tailored to the configuration of the GPU and etc., and a kernel without any optimization. Changing the size of a particular architectural parameter of a GPU may have some affects on other architectural parameters. For instance, increasing register usage of each thread in a kernel decreases the number of concurrent threads in each SM which degrades the memory hiding efficiency as there is less computation available. To obtain the high performance, the developer needs to try all combination of possible optimizations and consider the set of optimizations that result in the highest performance. This highly time consuming processes needs to be performed for each application.

Performance models help the GPGPU application developers to evaluate the ap-

[12] Application-Specific Integrated Circuit.

plication's performance, estimate the benefits of the possible optimization and find the performance bottlenecks of the application. Also, compilers can utilize the performance models to apply the most beneficial set of optimizations to the applications. In this section we review the literature on GPGPU performance modeling.

Zhang and Owens proposed a microbenchmark based performance model that allows programmers to determine the performance bottlenecks and potential performance optimizations and architectural improvements [100]. The proposed model considers three components of a GPU for modeling; the instruction pipeline, shared memory access time and global memory access time. By measuring the time that a program spends on each component, the model determines the program bottleneck as the component that takes most of the program's time.

To model instruction pipeline, instructions were categorized based on their costs. The cost was determined by the number of functional units in an SM that can execute the instruction. Then the model calculated the peak throughput of each instruction type. To estimate the throughput of the pipeline, micro-benchmarks that repeatedly ran an instruction of each type at different level of warp parallelism were used. Then, the execution time of a general program was estimated as a linear combination of the time spend on each instruction type.

To model the shared memory, a shared memory micro-benchmark was used which repeatedly transfers a block of data from one region of memory to another and measures the memory bandwidth at different level of warp parallelism. The results indicated that shared memory had a longer memory pipeline than the instruction pipeline and needed more warp parallelism to hide its latency. After running the shared memory micro-benchmark, the model estimated the time of shared memory by using the bandwidth at a corresponding level of warp-parallelism for a general application. Modeling the global memory was more complex due to its sensitivity to other factors such as the number of blocks and the number of threads per block. In order to estimate the global memory bandwidth accurately, a synthetic benchmark was generated for each general application with the same number of blocks, block size and memory transactions per thread.

After determining the performance bottleneck, the model provides information to track down the possible causes of the bottleneck. For example, for instruction pipeline, the low computational intensity, expensive instructions or insufficient parallel warp can cause the instruction pipeline to become the bottleneck. The programmer can use this information to find a programming or architectural solution to improve

the performance. The study of three GPGPU applications showed that the proposed performance model could capture the main GPU performance factors within an acceptable error of 5-15%.

Newsha Ardalani et al. developed a studied the performance gain that an application can achieve by being ported to GPU. They developed a machine learning based model that predict the performance of an application on a GPU based on the single-threaded CPU implementation of the application [2]. They trained the NN model using a large set of previously implemented GPU code along with their CPU implementation. The model learns the non-linear relationship between the GPU execution time and the features of the program. The model predicts the execution time of a new application by receiving its features such as total number of instructions, instruction type, number of cold misses, number of memory references with reuse distance of less than 2, etc. The model was evaluated with 10 benchmarks and achieved a relative error of 22.4% and geometric mean of 11.6%, with minimum error of 0.5% and maximum error of 54.6%.

Lopes et al. applied cache-aware Roofline modeling principles to model the upper-bounds for performance, power and energy efficiency of GPGPU applications [55].The original Roofline model [91] relates the performance of the CPU to the off-chip memory traffic. The model proposed by Lopes considers the the peak throughput of the execution units and different memory elements such as device memory, cache levels and shared memory to construct the models. The throughput upper-bounds are obtained through intensive micro benchmarking of 9 NVIDIA GPUs from 3 different generations. They evaluated the proposed models by using them to characterize 23 GPGPU applications in terms of the GPU resource that determines the performance upper-bound. The results show for most of the benchmarks L2 cache and device memory are the performance bottleneck.

Sim et al. proposed an analytical GPU performance model, called $GPUPref$, to help programmers identify performance bottlenecks and optimize the code [81]. $GPUPref$ quantitatively estimates potential performance gains by calculating inter-thread Instruction Level Parallelism (ILP), Memory Level Parallelism (MLP), computing efficiency and serialization effects and suggests the optimization that the programmer should try first.

Tools were developed to provide $GPUPref$ access to hardware performance counters, information such as instruction category or loop trip count and ILP and MLP information. These data are fed to the analytical model, that was part of $GPUPref$,

which is a refinement of the one presented earlier [37]. The analytical model estimates the execution time as the summation of computation time and memory access time while subtracting the overlapped time, which indicates how much of memory access time can be hidden by multithreading. The computation time is estimated considering the total number of instructions an SM executes, the instruction throughput and serialization overhead due to synchronization instructions, SFU[13] resource contention, control flow divergence and shared memory bank conflicts. The memory access cost is dependent on the number of memory requests, memory latency of each request and the degree of memory level parallelism.

The performance model can be used to provides the programmer with an insight to the program bottlenecks and potential solutions. It estimates the potential benefits by improving inter-thread instruction-level parallelism, memory level parallelism, removing inefficient computation and the overhead of serialization.

In [57], the authors reviewed different heterogeneous system performance modeling techniques. They categorized performance modeling approaches into analytical, statistical and machine learning based, quantitative and compiler-based methods.

Building analytical models require deep understanding of the hardware and the workloads. Once built they are fast to evaluate. Models built based on statistics and machine learning techniques use regression to obtain the parameters that impact the performance of the system the most and the model is built based on those parameters. Before using these models for performance evaluation, they need to be trained using a set of benchmarks for which the performance values are known. The training process can be time-intensive but while trained with a proper set of data, they produce accurate results.

In quantitative approach, a set of performance models are driven based on measurements performed using micro-benchmarking technique to exercise different components of the architecture under study and capture their effect on the performance. These models can be used to study performance bottlenecks [100].

Compiler-based methods are combination of kernel analysis and analytical modeling. They predict the performance and determine performance bottlenecks [4].

Dublish et al. studied memory bottlenecks of GPGPU applications [18]. The authors argued that GPUs' cache hierarchy partially filters bandwidth demand to off-chip memory due to massive computational and memory demand of GPGPU ap-

[13]Special Function Unit is responsible for executing unusual functions such as transcendental functions, interpolation for parameter blending, reciprocal, reciprocal square root, sine and cosine.

plications. Their analysis of three major memory system components, i.e. private L1 cache, shared L2 cache and off-chip memory, showed that the bandwidth bottlenecks cause high congestion in the memory system and leads to high latency in the critical path. The results showed that the performance improvement obtained by increasing the bandwidth of a single memory component in isolation can be sub-optimal compared to the performance improvement that can be achieved by combined increase of bandwidth of memory components. The reason is that the former creates congestion in other memory components while the later eliminates that.

1.4 GPU Design Space Exploration

Performance that a GPU delivers, depends on the design parameters which are determined at manufacturing time, such as the number of SMs, SIMD width, off-chip memory bandwidth, number of memory controllers, and some configuration parameters that can be set at compile time using primitives, such as the presence and the size of L1 cache, shared cache size etc. [68].

On the application side, the performance depends on applications characteristics such as the memory access patterns and computational intensity. Therefore the configuration that delivers a good performance for an application might result in a poor performance for another application. To find the best configuration for an application, one needs to explore the large design space of GPUs by simulating all possible configurations which usually is not feasible. There are few studies that propose methods to simplify the design space exploration. In this section we review these works.

Jia et al. proposed Stargazer, an automated GPU performance exploration framework based on stepwise regression modeling [42]. The method employs a stepwise linear regression to relate the system performance to a number of independent architectural parameters and their pairwise interactions.

To build the model, the proposed method starts with the first architectural parameter from a given set of independent parameters. The model is made using linear regression based on this single parameter and the method calculates the coefficient of determination (R^2)[14] to measure the quality of the model. Then the model is extended by adding another parameter. The calculation of the adjusted R^2 determines

[14] The coefficient of determination of a linear regression model is the quotient of the variances of the fitted values and observed values of the dependent variable. It is interpreted as the goodness of fit of a regression.

whether the new parameter brings more benefit to the model. Then the method adds the interaction between two parameters and evaluates the significance of the term in the model. The method repeats the same process for all the parameters under the study. After this stage, the model consists of the most influential parameters along with their pairwise interactions.

In the models we have developed, as we shall see in the following chapters, we have used fold-over Placket-Burman (PB) design [76] to evaluate the importance of different architectural parameters. Like the step-wise regression method, the fold-over PB design considers the pair-wise interactions between parameters [15]. The PB design requires the response of the model for a number of different configurations. These responses can be obtained via simulation. In PB design the number of simulations grows linearly with the number of parameters under the study. However, in the regression model used in the Stargazer [42], the computational complexity grows non-linearly as the number of pairs to be evaluated for inter-parameter interaction increases factorially. Therefore, for the models with a large number of parameters, PB design seems to be more efficient in terms of model development time.

A Least squares solver was used to solve the model and the coefficients were determined using a small set of configurations randomly chosen from the design space. The evaluation showed that using 1% of the configurations in the design space, the model could predict the performance of an application with less than 5% error. The results showed that SIMD width is the dominant parameter and block concurrency, number of shared memory ports and DRAM queue size come next.

Ryoo et al. studied the challenges of GPU applications optimization caused by interactions among the architectural constraints such as register file size, threads per SM, threads per thread block etc. [79]. The optimizations applicable to applications were considered as follows: providing enough warps to hide the stalling effect of long latency and blocking operations, redistribution of work across threads and thread blocks, reducing the dynamic instruction count per thread, intra-thread parallelism and resource balancing. As each of these optimization techniques usually affects different aspects of system behavior, selecting the best set of optimizations for each application that leads to the optimum performance is not a trivial task. To this end, performance metrics, i.e. efficiency and utilization, were derived from static code and used to prune the optimization search space. Efficiency was defined as the inverse of

[15] It is worth noting that the interaction between parameters may be more than pair-wise but PB design technique cannot capture more than pair-wise interaction between the parameters

the total number of dynamic instructions that will be executed by all the threads of the kernel. Utilization was calculated as the average number of compute instructions a warp executes before running into a blocking instruction, multiplied by the number of warps in a SM which can be executed while a blocking instruction in a warp is being resolved.

The proposed method uses two flags provided by NVidia Cuda Compiler (nvcc) CUDA compiler. *Cubin* flag was used to obtain the resource usage, i.e. the number of thread blocks per SM and the number of warps per thread block. *ptx* flag was used to determine the execution profile, i.e. the total number of instructions and the number of blocking instructions. These flags provide the information needed to compute the performance metrics.

In the proposed method, all the configurations in the optimization space were compiled with these flags and the performance metrics corresponding to each configuration were calculated. The metrics were not detailed enough to be used to determine the optimal configuration and were used to narrow down the space of possible configurations. To this end, the metrics values were plotted for each configuration and the high utilization and high efficiency configurations were selected as the Pareto Optimal subset. Using the proposed method, the optimization space was reduced up to 98%.

Dutta et al. [21] used different machine learning techniques to explore Dynamic Voltage and Frequency Scaling (DVFS) space and predict the power dissipation of GPUs in different DVFS state. Using the proposed prediction model, one can associate each phase of the execution of an application running on a GPU to a DVFS state that maximizes the performance while keeping the power dissipation below a predefined power cap enforced on the system. To build the model, the on chip power meter and performance counters of several architectural parameters of NVIDIA Quadro P4000 GPU are collected using the nvprof profiling tool. The authors have reported that among eight machine learning technique they have used for power prediction, a weighted average of the predictions made by Sequential Minimal Optimization (SMO) regression, simple linear regression and reduced error pruning tree showed the best accuracy. NeuralNet model performed the worst due to small training data set.

Yu et al. studied the importance of GPGPU architecture parameters with respect to performance and how these parameters interact with each other [98]. They used stochastic gradient boosted regression tree machine learning technique and trained their model using performance values obtained by running a set of GPGPU applica-

tions on a random set of GPU architectures. From the predictor results, they used an ensemble-learning based approach to quantify the importance and intensity of the interactions between different parameters. They evaluated the model by running 25 GPGPU applications and showed the core frequency and the maximum number of thread blocks per core are the most dominant parameters. Some benchmarks were sensitive to L1 data cache size and interconnect frequency. The results indicated that most of the benchmarks were sensitive to 2 to 5 pairwise parameter interaction.

Jia Wenhao et al. studied the challenges in tuning power and performance of GPGPU applications [41]. They developed a regression tree based model to study the design space of GPGPU applications and accelerate the tuning process. The proposed model uses a small portion of the design space to recursively partition a given design space at key inflection points of a design parameter values. The parameters are application specific. A few examples of the parameters used for different benchmarks include number of threads in each thread block, the dimensions of each thread block and the number of elements processed by each thread. To predict the performance of a new application, the application parses the nodes of the tree starting from the root and follow the branches based on how the parameters of the new application compares to the division values of the branch. When the application reaches the leaf, the average power and performance of sampled designs in that partition are used to make the predictions. The authors reported average error of 4% and 8% for power and performance predictions, respectively.

1.5 Neural Networks

Neural Network (NN) is a machine learning technique inspired by human brain. Human brain is composed of billions of neurons. Each neuron has, on average, 7,000 synapses connection to other neurons. To learn complex and non-linear hypothesis, we can construct a model using much less neurons and connections between them and still achieve accurate results.

The simplest form of a NN is a single-layer, single-neuron model or a *Perceptron* shown in Figure 1.7. It has a set of inputs, and based on the inputs and a bias value it produces an output (i.e. 0 or 1). The wights indicate the strength of the input values. The combination function calculates the weighted inputs and adds the bias value to it. The activation function receives the weighted sum of the inputs and the bias value, maps it to a certain range (e.g. 0 to 1) and compares it with a threshold

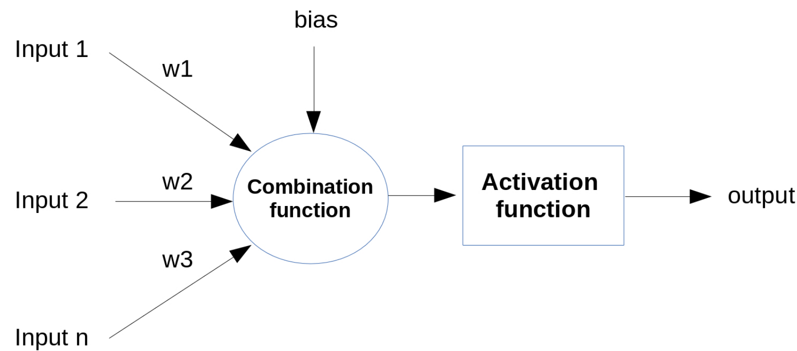


Figure 1.7: A single-layer neural network structure [77].

and sets the output (activation) value. The bias value is used to shift the activation function to the left or right. Equation 1.1 summarizes the perceptron calculation.

$$y = g(\sum_{i=1}^{N} w_i * x_i + bias) \tag{1.1}$$

in which y is the perceptron's output, g denotes the activation function[16], N is the number of inputs, w_i is the weight associated with input i, x_i is input i. For the one perceptron NN model shown in Figure 1.7, a simple linear activation function can be used to learn simple problems (e.g. to implement AND logic). To learn the behavior of more complex problems, a more sophisticated NN structure and a non-linear activation function needs to be used. By using a non-linear activation function we can establish a non-linear relation between the output of the system and the inputs which is essential to learn the behavior of complex problems.

To train the perceptron to learn the behavior of a function (e.g. an AND gate), we start with a set of initial weights and feed the inputs to the percepteron. If the output is not the expected value, the weights are updated using the *delta rule*. It determines the amount of the changes in the weights of the network based on the difference between the expected and the obtained output of the network multiplied by a learning rate [23]. If the output of the network is the expected value, the weights stay unchanged. If the output is larger or smaller than the expected value the weights decreased or increased, respectively. The learning rate controls how much we change weights at each step. A very small learning rate makes the algorithm very slow whereas a learning rate that is too large may make the algorithm never converge. After we update the weights, the process repeats until it produces the correct results

[16] Activation function, in general, is a bounded non-decreasing nonlinear function [77].

for every possible combination of input. A perceptron can be used as a linear binary classifier. However, to model the behavior of more complex problems, a multilayer perceptron model should be used.

Figure 1.8 shows an example of a multilayer NN. It consists of an input layer, one or more hidden layers with variable number of neurons in each layer, and an output layer.

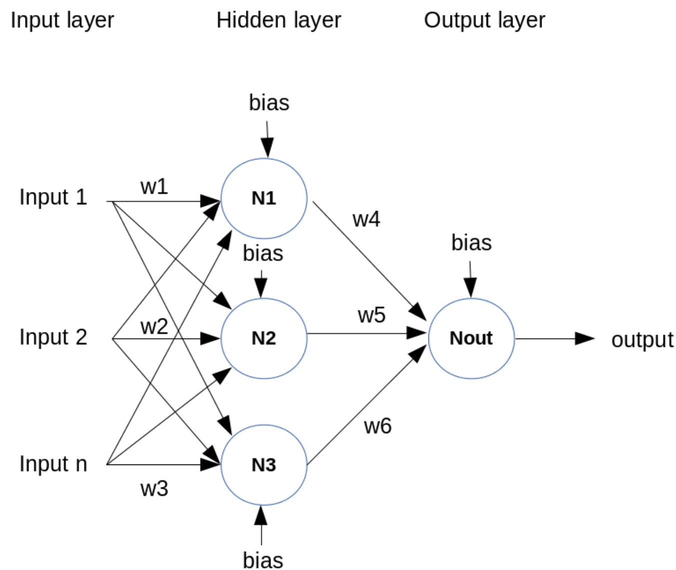


Figure 1.8: A multilayer neural network with one hidden layer with three neurons [77].

To produce an output for a given input vector, the activation value of each node of the hidden layer are calculated using equation 1.1. The activation values generated by neurons in the hidden layer are the inputs of the next layer and they pass forward to the next hidden layer or the output layer which produces the final result.

The training algorithm for multilayer networks is called *back propagation*. It is a form of *Gradient Descent (GD)* [77], but accounts for the fact that the derivatives of the error function with respect to the hidden weights cannot be evaluated directly and easily because of the non-linearities an layaring. The error is formulated as a cost function that can be minimized using different optimization techniques such as Gradient Descent. Detailed description of GD algorithm can be found in Chapter 10 of [77].

For the rest of this section we present the terminology and definition of some of the techniques that are used in the NN model we developed and we are going to refer to in the following chapters.

Supervised learning: In supervised learning, the NN model is built based on a set of samples for which the expected output values are known. The known set is called the training set and is used, during training phase, to adjust the weights of the network. Once the model is trained, it can predict the output for unseen inputs.

Feedforward Networks: A neural networks with no feedback loop from its output to its inputs is called a feedforward neural network. The structure shown in Figure1.8 is an example of a feedforward neural network. Feedforward NNs are commonly used for supervised learning.

Hidden Layer Size: The structure of our NN model is composed of only one hidden layer. The number of neurons in the hidden layer depends on the complexity of the problem and is application specific. We examine hidden layer sizes 2 to 15 and train 500 NNs of each size and select the hidden layer size with the smallest mean squared error.

Training Process: A process in which the model learns the relationship between the inputs and the target outputs of the model. The training process is a repetitive process that tries to minimize the cost function using an optimization algorithm and adjusting weights until the cost function, i.e. error of the network in predicting the target output of the known exemplars, is minimized.

Over-Fitting: The problem of NN model fitting to the training set too closely but failing to generalize and make accurate predictions for new exemplars is called over-fitting problem.

Training, Test, Validation and Blind Sets: In neural networks, the set of exemplars that is used to construct the model is divided into three sets; training set, validation set and test set. The training set is used to adjust the weights of the network using back-propagation algorithm. The validation set is used to prevent over-fitting by discarding the NNs that perform well on the training set but produce large error on the validation set. Once the model is trained, the test set is used to test the performance and generalization capability of the model, meaning that how well the model predict the output of exemplars that it has not seen before. The blind set is a set of exemplars that we want to predict their output using our trained NN mode.

Regularization: Regularization is one of the techniques to overcome over fitting. It improves the generalization capability of the model. For example, one of the methods used to regularized the model is to make the model simpler by penalizing the weights by adding new terms to the cost function which results in smaller weights

at the end of the training phase.

Ensemble NN: When a number of neural networks are trained and a combination of their outputs are used to produce the final result we call it neural network ensemble. It helps to improve the generalization capability of the model. We use an ensemble NN model to predict the power and performance of GPGPU applications.

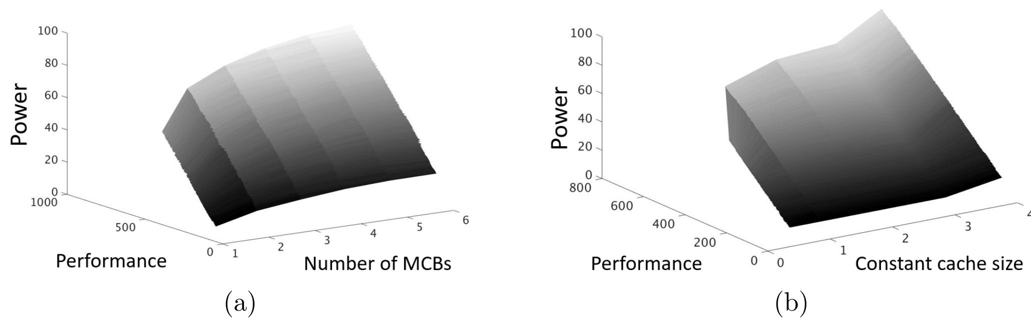
Chapter 2

Introduction and Thesis Roadmap

Application developers that are looking for speeding up the execution of their applications, may find GPUs as a good choice. Especially, with recent developments in GPGPU programming languages, such as CUDA [65, 68], OpenCL [30] and OpenAcc [70], the time and effort they need to put in creating a version of their applications that can be run on a GPU is reduced considerably. However, using a GPU as an accelerator is beneficial only if the application could utilize the GPU's compute resources and memory bandwidth. To do so, the application should have a large amount of compute intensive parallelism and memory behavior that can be overlapped, and therefore hidden, by computations.

To achieve this, the application developer needs to study the attributes of the application and be familiar with the target GPU architecture. Thereafter, they can decide whether running the application on the GPU is beneficial or not. If GPU seems to be a good choice, how efficient the application execution on the target GPU would be is a harder question to answer and depends on how well the application's compute and memory requirements are balanced with the resources available on the target GPU.

Figure 2.1 shows the power and performance of AES [58] benchmark's design space running on a Fermi-like GPU. The surface on this figure is composed of more than 23,000 configurations each of which represents a different GPU configuration in terms of the size of memory components such as caches, Memory Controller Block (MCB)s and register files. The figure is a "projection" of the 7-dimensional design space to (a) the number of memory controller blocks and (b) the constant-cache size. Both figures illustrate the complex relations of power and performance to the GPU configuration. The top right corner of the surfaces represents the configurations with the highest



(a) (b)

Figure 2.1: Projection of the design space to (a) number of memory controller blocks and (b) constant cache size.

performance and power dissipation. The corner closest to the origin, represents the configurations with the lowest performance and power dissipation. As it can be seen the power and performance are limited by characteristics of the applications as well as available GPU resources.

Each application has its own power and performance requirements. We have developed models that link the application with the power and performance. The models predict the power-performance optimum configurations. Then, we extend these models to identify configurations that behave ”well” over many different applications. As different applications have contradicting requirements, we use the term ”well” to indicate that on the average, all applications benefit from the chosen configuration.

As we discussed in Chapter 1, Section 1.3, the power and performance of an application depends on the characteristics of the application as well. For example, memory access pattern, compute or memory intensiveness and the level of the parallelism are some of the factors that can limit the achievable performance of the application. We developed a model that predicts the optimum GPU configurations for GPGPU applications based on the static characteristics of the applications. We proposed joint power-performance optimization techniques and other heuristics to improve the accuracy of the proposed models.

We have evaluated the proposed models with a set of GPGPU benchmarks. The results show that the proposed models predict the optimum power-performance configurations for most of the cases. For the other cases, the power and performance of the model predicted configurations are very close to the optimum configurations. We could also successfully predict the power-performance configurations based on the characteristics of the applications.

In this thesis we were interested to answer the following questions.

1. **Given a GPU configuration, can we predict its power and performance when we run an application on it?**

   To answer this this question we proceeded as follows.

   - We studied the design space and attributes of 24 GPGPU applications. We introduce the benchmarks in Section 2.1.
   - W showed that GPU's performance is linked to its architectural parameters. We introduce a fast, low-cost and effective approach to optimize resource allocation in future GPUs [47]. Section 2.2 introduces this approach.
   - We develop a NN-based model to predict power and performance of GPGPU applications [44]. We present more details in Section 2.3.
   - We propose to use a multi-objective optimization technique to obtain a subset of the design space that contains configurations that are power-performance optimum [45]. We present more details in Section 2.4.

2. **Given the characteristics of an application, can we predict the GPU configuration that runs the application optimally in terms of power and performance?**

   - We develop a model to predicts the power-performance optimum GPU configurations based on code attributes [46]. We introduce this model in Section 2.5.

3. **Given a set of GPU applications, can we predict the GPU configuration that performs well over all the applications?**

   - we introduced a method to predict the GPU configuration that all the applications benefit from running on it. Section 2.6 briefly introduces this method.

We have improved some of the modeling techniques as follows.

- We introduce an outlier detection heuristic to capture predictions with large errors [49]. Section 2.7 briefly introduces this method.

- We propose an alternative to a heuristic used in the NN model that chooses the validation set for training purpose and propose to the alternative with mathematical solvers and Quantum Annealing technique. This alternative speeds up the training process significantly [48]. Section 2.8 briefly introduces this method.

In the following sections, we will expand on the contribution statements and introduce the techniques involved and the results obtained.

2.1 Benchmarks, Simulator and GPU Architecture

To evaluate the proposed models, we have used applications from two benchmark suites, i.e. CUDA SDK Sample Codes [66], Rodinia benchmark suite [8] and some other benchmarks that are widely used in GPGPU research area for power and performance evaluation [1, 6, 28, 58, 60, 75, 86, 86].

We have evaluated different models with the benchmarks reported in Table 2.1. These benchmarks vary in terms of compute and memory demands, number of kernels and level of parallelism. In choosing the benchmarks we were limited to the ones that were compatible with the GPGPU simulator we use to validate the predictions made by our models. Table 2.1 summarizes the benchmarks and their characteristics. We will present more details about the benchmarks in the following chapters.

As many of the GPU architectures we explore are not implemented in silicon, we use GPGPU-Sim simulator and configure it to implement different GPU architectures. GPGPU-Sim is a cycle-level GPU power and performance simulator developed at the University of British Columbia [5]. The latest GPU architecture GPGPU-Sim supports is NVIDIA's Fermi architecture. Therefore, we have used Fermi GPU as our baseline GPU architecture as reported in Table 2.2. There are seven memory related parameters not shown in the table which vary in different GPU architectures that the models we propose explore their sizes to achieve application specific optimum power-performance GPU architecture.

Table 2.1: GPGPU Benchmarks used to evaluate the proposed models.

benchmark	description
NN [6]	Neural Network
LPS [28]	3D Laplace Solver
STO [1]	SoreGPU
CP [86]	Coulombic Potential
AES [58]	Cryptography
RAY [60]	Ray Tracing
BFS [8]	Breadth first search
NQU [75]	N Queens Solver
Needle [8]	DNA sequence alignments
HotSpot [8]	Thermal modeling
BackProp [8]	Back propagation
SRAD [8]	Speckle Reducing Anisotropic Diffusion
VecAdd [64]	Vector addition
Gaussian [8]	Linear system solver using Gaussian Elimination
DWT [64]	Sparse Matrix Collection
Transpose [64]	Transposing a floating point matrix
EigenValues [64]	Computation of eigenvalues
ConvolutionTexture [64]	A separable convolution filter of a 2D image without using shared memory
MergeSort [64]	Merge Sort
ConvolutionSeparable [64]	A separable convolution filter of a 2D image
LIB [86]	Swaption portfolio calculation
MUM [8]	High Throughput Sequence alignment
ScalarProd [64]	Scalar products of input vector pairs
Histogram [64]	64bin histogram calculation

2.2 Design Space Exploration for Efficient Resource Allocation

As technology advancements provide designers with a growing chip real-estate, effective methods of exploiting the available resources becomes essential. The goal of this work is to revisit GPU design and introduce a fast, low-cost and effective approach to optimize resource allocation in future GPUs. Our solution utilizes the Plackett-Burman (PB) methodology [76] and formulates a constraint optimization problem to find the optimal GPU design for different available chip real-estate budgets without resorting to exhaustive design space exploration. Our results show that for the applications and configurations investigated, our solution matches the design suggested by an exhaustive search for most of the cases, and for the other cases the proposed

Table 2.2: GPU configuration.

Parameter	Value	Parameter	Value
Number of shaders	30	Warp width	32
Shader clock frequency	1.3 GHZ	Scheduling	PDOM
Max thread per shader	1024	max CTA/Shader	8
SIMD pipline width	32		

solution is the second best design.

2.3 Design Space Exploration using a NN-Based Power and Performance Predictor

In the model presented in Section 2.2, we assumed that the effect of changing the size of each architectural parameter on performance is linear. This is true only for a small sub-range of the effective range of each parameter. Especially, when the performance is becoming saturated, the performance gain from having more of those units on the GPU becomes smaller. For example, consider a memory intensive application for which we change the number of memory controller blocks. The performance of the benchmark may double if we change the number of memory controller blocks from 1 to 2, and from 2 to 4. But the performance may not be as sensitive when we increase it to more than 4 memory controller blocks because of the memory access characteristics of the application. Considering only the variation of the parameter that linearly affects the performance reduces the solution space that the model searches for the optimum result. However, the optimum result may not be in this short version of the design space. Also, Plackett-Burman design does not account for intera-parameter interactions.

If we want to consider the full effective range of each architectural parameter, then we need a model that learns this non-linear behavior. We propose to link application power and performance to GPU configuration. To this end, we use a neural network based predictor to learn the relationship between the power and performance of GPGPU applications and the configuration of the GPU. The advantage of a NN-based predictor is its ability to learn the non-linear effect that changing the size of architectural parameters has on the system's response, i.e. power and performance.

For each benchmark we create a pool of GPU configurations by varying the size

of seven architectural parameters such as, caches, register file and the number of memory controller blocks. Then, we simulate the execution of the benchmarks on different configurations to obtain the power and performance values. The size of the configuration pool, i.e. the design space, depends on benchmark's characteristics.

For each benchmark we build two models, one for power and one for performance. We train the models with a small subset of the design space and use the trained models to predict the power and performance of the rest of the design space.

We have evaluated the proposed models using GPGPU benchmarks listed in Table 2.1, with different design space sizes. We were able to achieve acceptable generalization ability of the model , indicated by low prediction errors, for each benchmark by utilizing a small subset of the design space for training. In order to determine the accuracy of the model, we compared the outputs of the model against results obtained from simulation. The trained models could successfully predict the activity of the configurations in the design space with acceptable error.

2.4 Predicting Optimum Configurations Using Multi-Objective Optimization Method

The NN-based power-performance predictor presented in Section 2.3 predicts, for a given application, the power and performance of all configurations in the design space. Based on the goal of the system, one may want to choose a subset of the design space that meets a certain criterion. For example, configurations that offer the highest performance with minimum power dissipation. To do so, one needs to examine the power and performance values of all configurations in the design space and find the power-performance optimum configurations.

Searching the whole design space is not a trivial task as for some benchmarks there are thousands of configurations needed to be explored. Therefore, we propose to augment the model presented in Section 2.3 by employing a multi-objective optimization technique.

We propose to use Pareto Optimal multi-objective optimization method [17] to obtain the optimum configurations with respect to multiple objectives. The objectives we have defined in this study are maximum performance and minimum power dissipation.

However, we are using the NN model to obtain the required power and performance

values to use in the Pareto Front optimization. At the end, given that the model is not error free, we obtain the true (simulated) power and performance values for the identified configurations and show that the obtained configurations are very close in power and performance from the actual Pareto Front. As such, our proposed method is accurate enough and results in semi-optimal configurations. Additionally, it is computationally efficient as it utilizes the NN model and it only needs to simulate a very small number of configurations.

2.5 Predicting Power-Performance Optimum Configuration Using Code Attributes

The model we presented in Section 2.3 uses an application specific NN-based predictor to explore and predict the design space of GPGPU applications. Although the proposed model produces accurate results, the training time of the model grows with the size of the design space. For applications with large design space the training phase of the model takes days to complete. Also, to train the model the actual power and performance values of the configurations in the training set are needed. We obtain actual power and performance values using a simulator as we discussed earlier in this chapter. Depending on the complexity of applications' behavior and the size of the design space, the training set may include hundreds of configurations and simulating those configurations can take a long time.

We propose to model GPU architectural parameters as a function of attributes of an application and hence obtain an application independent model. The model, when built and trained, can predict the power-performance optimum configurations for any given application based on the attributes of the application. To this end, we need to train the model using a set of applications for which we have the power-performance optimum configuration. The model we will present in Section 2.4 can be used for this purpose.

In the training phase, the inputs of the model are GPGPU kernel attributes, such as the grid size, thread block size and number of basic blocks and the optimum GPU configurations for the applications in the training set. Once the training phase is finished [1] the model can predict a power-performance optimum configuration based

[1] The time requires to train a model depends on the complexity of the problem and may vary from few hours to few days.

on application's attributes.

The advantage of this model over the model presented in Section 2.4 is that obtaining the attributes of an application is much faster than simulating the training set configurations. Although the user needs to obtain the power-performance optimum configurations for the applications in the training set, it is a one time task, and once done, the model can be used to predict unseen applications' optimum configurations.

2.6 Obtaining a Good Configuration for a Set of Application Code

As we discussed in Section 2.4, we use Pareto Front optimization technique to obtain a set of application-specific power-performance optimum configurations. We propose a method to obtain a GPU configuration that performs well for a set of applications with respect to variety of performance metrics such as high performance, low power and optimum power-performance.

2.7 Outlier Detection Heuristics

The model presented in Section 2.3 produces accurate results. However, there are some predictions with large errors (outliers). Although outliers comprise a small portion of the design space, it is desirable to determine and exclude them from the output of the model. We have devised methods to detect and filter out outliers using the characteristics and geometry of the design space as well as the process of developing the NN model.

2.8 Accelerating NN Ensemble Using Quantum Annealing

Although the NN model presented in Section 2.3 is a robust model, the model development phase takes a long time[2] due to the heuristics that it uses to prevent over-fitting and improve the generalization capability of the network. The heuristic that is used to prevent over-fitting is called Guided-selection heuristic.

[2] For some benchmarks it takes several days depending on the computational platform used for training.

In our model, the configurations in the set that is used to develop the model during the training phase is divided into training set, test set and validation set. Validation set is used to prevent over-fitting.

The Guided-selection heuristic is used to select the validation set. In the validation set we strive not to have exemplars that are unique and dissimilar from the ones in the training set. The Guided-selection heuristic determines how unique are the various exemplars by attempting to train NN without the said exemplar and determining the generalization capabilities of the resulting models. It uses this information to select a validation set that results in maximum generalization.

The method, because of the training requirement, is computationally expensive. We propose an alternative faster method that judges the suitability of an exemplar to be placed in the validation set based on its "distance" from the exemplars in the training set. If the "distance" is small, it could be placed in the validation set. We have formulated the problem as a quadratic optimization problem and used quantum annealing infrastructure to solve. The results show radical improvement in the development time of the model, while preserving the generalization capability of the NN model.

In the chapters that follow, we will present each of these models and refining heuristics in more details.

Chapter 3

Efficient Resource Allocation in GPUs

In the previous chapter, we briefly introduced the models we have developed to study the resource allocation, power and performance prediction, design space exploration, outlier detection and optimization techniques used to predict the optimum GPGPU configurations and accelerating training phase of the NN model. In this chapter, we present details of the model we developed for efficient resource allocation in future GPUs.

Employing Graphics Processing Units (GPUs) as accelerators for general-purpose throughput-intensive workloads has become an important part of high performance computing. As technology advancements provide designers with a growing chip real-estate, choosing a particular GPU part or parts (e.g. registers, caches) on which we would spend the limited transistor budget that result to a maximum improvement of a targeted performance indicator becomes challenging. The goal of this work is to revisit GPU design and introduce a fast, low-cost and effective approach to optimize resource allocation in future GPUs. Our solution utilizes the Plackett-Burman methodology [76] and formulates a constraint optimization problem to find the optimal GPU design for different available chip real-estate budgets without resorting to exhaustive design space exploration. Our results show that for the applications and configurations investigated, our solution matches the design suggested by an exhaustive search 48 out of 57 times, and for the cases the proposed solution did not match the one found by the exhaustive search, the maximum error is less than 3.5%.

The remaining of this chapter is organized as follows. In Section 3.1 we review

the Placket-Burman design and the Knapsack Problem that are used to develop the model. In Section 3.2 we discuss our methodology. In Section 3.3 we present the results, and review the related work in Section 3.4. We conclude with Section 3.5.

3.1 Background

3.1.1 Plackett Burman Design Technique

In design space exploration, one needs to conduct numerous experiments varying a number of parameters to ensure full exploration. In designs with N parameters, each one taking one of L values, L^N experiments need to be conducted. Plackett and Burman [76] introduced a method which can measure the effects of the N parameters, where multi-parameter interaction is not present, by conducting X experiments, where X is the next multiple of 4 strictly greater than N. A Plackett-Burman design with fold-over, can quantify the effect of two interacting parameters. To capture the pairwise interaction of the parameters [96] we have utilized a Plackett-Burman design with fold-over.

Plackett and Burman [76] introduced templates of experiments for different number of parameters. Table 3.1 shows an example of such a template (with fold-over). The rows correspond to the experiments while the columns correspond to the parameters. A "+" ("−") sign indicates that for the experiment indicated, the corresponding parameter is set to its maximum (minimum) value.

For each experiment i, a target performance metric T_i is measured. After running all experiments, it is possible to calculate the effect of each parameter on the design. We denote by S_α the effect of parameter α on the target performance metric. S_α can be calculated by subtracting the target values corresponding to the low parameters from those when they are high. For example, from Table 3.1, the effect S_E of parameter E is calculated as:

$$S_E = T_1 - T_2 + T_3 + T_4 + T_5 - T_6 - T_7 - T_8 - T_9 + T_{10} - T_{11} - T_{12} - T_{13} + T_{14} + T_{15} + T_{16}$$

The larger the magnitude the more influence the parameter has on the target performance metric. We are using this in the techniques introduced in this chapter as well as in deriving an additional descriptor of the NN models introduced in Chapter 4.

Table 3.1: Placket-Burman design with fold-over; 7 parameters, 16 experiments.

experiment	parameters							Target
	A	B	C	D	E	F	G	performance
1	+	+	+	-	+	-	-	T_1
2	-	+	+	+	-	+	-	T_2
3	-	-	+	+	+	-	+	T_3
4	+	-	-	+	+	+	-	T_4
5	-	+	-	-	+	+	+	T_5
6	+	-	+	-	-	+	+	T_6
7	+	+	-	+	-	-	+	T_7
8	-	-	-	-	-	-	-	T_8
9	-	-	-	+	-	+	+	T_9
10	+	-	-	-	+	-	+	T_{10}
11	+	+	-	-	-	+	-	T_{11}
12	-	+	+	-	-	-	+	T_{12}
13	+	-	+	+	-	-	-	T_{13}
14	-	+	-	+	+	-	-	T_{14}
15	-	-	+	-	+	+	-	T_{15}
16	+	+	+	+	+	+	+	T_{16}
	S_A	S_B	S_C	S_D	S_E	S_F	S_G	

3.1.2 Knapsack Problem

Given a transistor budget to spend on a GPU's resources, we need a method to choose the architectural parameters to spend the transistor budget on that result to a maximum improvement of a targeted performance indicator.

Using the PB design method, we have obtained the importance or value of each architectural parameter. There is a transistor count associated to each parameter that defines its cost (See Section 3.2.1). For a given transistor budget, the model tries to find the GPU configuration that fits within the budget and maximized the performance. This best fits the Knapsack problem [88].

Denoting by v_j (w_j) the value (weight) of an item of *typej*, b_j the maximum number of items of type *typej* and C the capacity of the knapsack, the knapsack problem is stated as:

Select x_j *(j = 1, ..., n)* items of *typej* so as to maximize $z = \sum_{i=1}^{n} v_j x_j$
subject to $\sum_{j=1}^{n} w_j x_j \leq C$; $0 \leq x_j \leq b_j$ and integer; $j \in N = 1, \ldots, n$

This is a Bounded Knapsack Problem (BKP) [88], a generalization of the 0-1 knapsack problem. We assume that v_j, w_j, b_j and C are positive integers. We used Knapsack optimization technique to optimize the resource allocation under different transistor budget scenarios. A sample code can be found in the Appendix A.1.

One can solve Knapsack problem using brute force method by trying all possible solutions in the design space which is not computationally efficient. Other methods are exact methods [19], such as branch-and-bound and linear programming and sub-optimal and probabilistic methods [24]. We solve the Knapsack problem using integer linear programming method [19] as it produces exact solution and the solver is available through Matlab's optimization package.

3.2 Methodology

Imagine we are designing the next generation of a GPU and because of a larger chip or smaller technology size, we can have 300 million more transistors on the chip. On which architectural element should we spend them to maximize the performance improvement of the new GPU?

The objective of the model we propose in this chapter is to find the GPU configuration that achieves maximum performance under a given transistor budget. Such problem can be solved by an exhaustive design space exploration, which is computationally expensive.

Our approach formulates the problem as a knapsack problem which can be solved efficiently [73]. In formulating the knapsack problem, one needs to identify the *value* (v_j) and *weight* (w_j) parameters. While the *weight* parameter is easily identifiable as the transistor count of each unit, the *value* parameter is not as directly identifiable. The *value* parameter simply shows the contribution of each of the architectural units to the performance of the overall architecture and for a particular application. This information is not readily available. We postulated that within the domain of exploration, the contribution of each unit to the performance is proportional to its cardinality and or size. The proportionality coefficient (i.e. the *value*) though is unknown. In this work, we shall show that the effect S_α of parameter α as per Plackett-Burman, can be used as the *value*.

3.2.1 Establishing the transistor count

Configuring a GPU involves choosing the number of memory controllers, the size of the data cache, the size of the register file and the size of the constant cache among other parameters. We included the parameters that had significant influence on the performance and we could estimate the transistor count of the parameters. For some of the parameters, e.g. SIMD cores, it was not possible to estimate the transistor count as there is not much information on the details of those parameters. However, the method we are presenting can be scaled with any number of parameters.

We estimated the cost (in number of transistors) for the data cache, the constant cache and the register file following the method cited in [82]. The method estimates the transistor count of memory-based components by accounting for the size of the component, the number of transistors required for each SRAM bit cell (4 transistors), each read (1 transistor) and write ports (2 transistor) of the bit cell.

For the cost of the memory controller, as the NVIDIA data are proprietary, we used the cost of a Xilinx memory controller core [95] as an estimation to NVIDIA's memory controller. However, the Xilinx design of the memory controller may not be efficient since it uses FPGA instead of custom silicon. The memory controller core supports SDRAM/DDR/DDR2/DDR3 memory. A DDR3 version of the core can be implemented on Spartan-6 LXT [93] to achieve the technology size (i.e. 45 nm vs Fermi's 40nm) and clock speed (i.e. 533 MHz vs Fermi's 700 MHz clock) close to the Fermi. The implementation requires approximately 300K transistors [95]. The cost for each unit are shown in Table 3.2.

Table 3.2: Transistor cost (in millions of transistors) of resource units

Parameter	Size/Number	Cost	Parameter	Size/Number	Cost
memory controller	1	0.3	Constant cache	32KB	52
DL1 cache	32KB	87	Register file	32KB	170

3.2.2 Benchmarks

We used the following set of GPGPU benchmarks from NVIDIA's CUDA Software Development Kit (SDK) [64] and Rodinia [8] benchmark suits: AES Cryptography (AES); Fast Walsh Transform (FWT); LIBOR Monte Carlo (LIB); 3D Laplace Solver (LPS); Montcarlo; Neural Network Digit Recognition (NN); Scan; Srad; Black sc-

hole; Hotspot; Ray tracing (Ray); Matrix multiplication (Matrix); Back propagation (Backprop).

These benchmarks were compiled and run on the GPGPU-Sim [5] simulator with the system configuration shown in Table 3.4. The ranges of the sizes or cardinality of the units used in configuring the GPU for our experiments, are shown in Table 3.3. The variation in the values of these parameters resulted in a large number of possible configurations. Totally, we experimented with 553 configurations. The simulations resulted in performance estimates for each configuration and for each benchmark, reported as Instructions Per Cycle (IPC).

Table 3.3: Benchmarks and low and high values of resources

Benchmark	MCB	DL1 Cache	Const Cache	Register File
AES	1-3	1KB,2KB	512B-8KB	4KB,8KB
FWT	1-4	4KB-32KB	N/A	4KB,8KB
LIB	3-5	32KB-128KB	64KB-256KB	2KB,4KB
LPS	1-3	1KB,2KB	N/A	2KB,4KB
Montcarlo	1-5	32KB,64KB	1KB,2KB	8KB,16KB
NN	1-4	8KB-32KB	512B,1KB	4KB,8KB
Scan	1-3	512B-2KB	N/A	2KB-8KB
Srad	1-6	2KB-256KB	N/A	4KB-16KB
Blackschole	1-4	2KB-8KB	N/A	4KB,8KB
Hotspot	1,2	1KB,2KB	N/A	8KB,16KB
Ray	1-3	1KB,2KB	1KB-32KB	8KB,16kB
Matrix	1-5	1KB,2KB	N/A	4KB,8KB
Backprop	1-3	1KB,2KB	N/A	4KB,8KB

3.2.3 Use of Plackett-Burman and optimization

In order to determine the *value* parameters, we applied the Plackett-Burman methodology with fold-over for each benchmark. The number of parameters used was four (i.e. number of memory controllers, the size of the data cache, the size of the register file and the size of the constant cache). These parameters have significant influence on the performance and we could estimate their transistor count.

The resulting effect values, S_α (one for each of the configuration parameters), are used as the *value* parameters for the knapsack optimization problem while as the *weight* parameters we used the transistor counts for each of the units as discussed in Section 3.2.1 above.

Table 3.4: GPU configuration.

Parameter	Value	Parameter	Value
Number of shaders	30	Warp width	32
Shader clock frequency	1.3 GHZ	Scheduling	PDOM
Max thread per shader	1024	max CTA/Shader	8
SIMD pipeline width	32		

We postulated earlier that the contribution of each of the units relates linearly to the performance obtained. Figure 3.1 shows the performance (as IPC) as a function of the number of memory controllers and for different sizes of the Data Level 1 (DL1) cache for the LIB benchmark. As the figure shows, the linear dependence of the performance on the number of controllers is evident for most of the domain of exploration.

We used Matlab's[1] linprog function to solve the knapsack problems. The linprog a linear programming problem and accepts real solutions, we imposed a mapping from the real solution space to an integer discrete space.

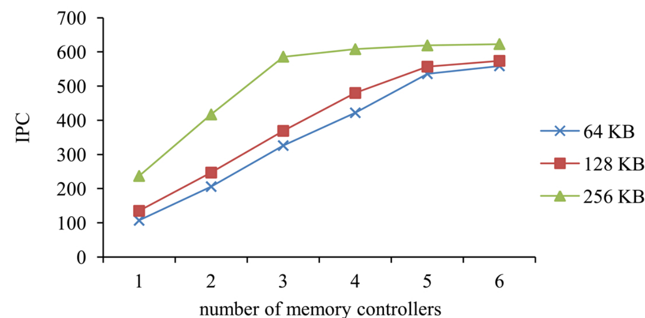


Figure 3.1: Performance of LIB benchmark for different sizes of DL1 cache and number of memory controller blocks

3.3 Evaluation and Results

In this section we present the results. As mentioned earlier, in this model we have studied four GPU resources as follows: memory controller block, DL1 cache, constant cache and register file. Table 3.2 shows the estimated costs of the parameters in terms of transistor counts while Table 3.3 shows the ranges of the configuration parameters.

[1]Matlab is a registered trademark of The Math Works, Inc. linprog is in the optimization toolbox.

Note that not all benchmarks have constant memory access instructions. Therefore, we have evaluated our method using three or four parameters depending on the benchmark under simulation. We have limited the number of memory controller blocks (MCB) to six, as more than five MCBs do not affect performance for most benchmarks.

We run the optimization problem for a gradually increasing transistor budget and for each budget, we obtain the resulting optimum-performance configuration. Figure 3.2 shows the resulting configurations for the NN benchmark. The horizontal axis refers to the transistor budget while the vertical axis shows the number of units corresponding to the optimum configuration. In Figure 3.2, one can distinguish regions of the transistor budget where the configuration does not change. For the NN benchmark (please refer to Figure 3.2) we distinguish five regions [2] marked by numbers 1 to 5 on Figure 3.2.

As an example, for a transistor budget of no more than 67 million transistors (i.e. within region 2), the optimum configuration comprises four memory controller blocks (MCBs), 2 units of DL1, one unit of Constant Cache and one unit of Register file. Some explanation is necessary here as to the definition of a unit. While the memory controller blocks are self evident, the units of the other parameters vary with the benchmark. For example for the NN benchmark (please refer to Table 3.3), a unit of DL1 Cache is 8KB, a unit of Constant Cache is 512B while a unit of Register File is 4KB. In this work, we have used as a unit the minimum size considered for each configuration parameter and for each benchmark, and then allowed the optimization function to return as a solution a configuration comprising multiples of the units of each parameter. For the said example, the optimum configuration obtained comprises 4 Memory Controller Blocks, 16 KB of DL1 Cache, 512B of Constant Cache, and a 4KB register file. Please note that the caches and register files can only be configured in sizes that are power of two. If the solution of the optimization returned a configuration that had a parameter that violated the "power of two rule", then the size of this parameter was fixed to a power of two that was less than the size suggested by the optimization, the excess transistor budget was returned to the transistor budget, and another optimization was run on the remaining parameters.

To ensure that the configurations suggested by the optimization method are indeed optimal, we also performed an exhaustive search of all possible configurations adhering to the transistor budgets considered and compared the results. We shall analyze these

[2] We define a region as the range of transistor budget for which the configuration does not change.

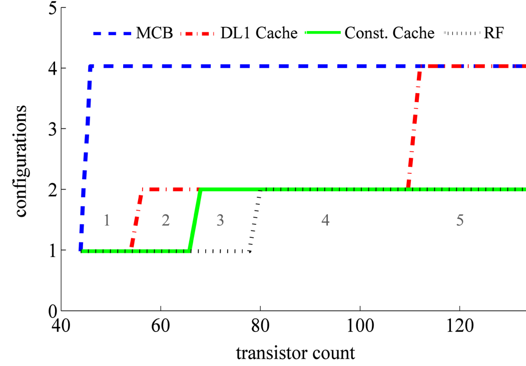


Figure 3.2: Optimum configuration regions for the NN benchmark.

results in Section 3.3.1 below. To conclude the example of the NN benchmark, Table 3.5 presents all possible configurations that adhere to the transistor budget of 67 million. The configurations are obtained by enumerating the solutions. As it can be verified, the configuration suggested by the optimization solution (Conf12) has indeed the optimum performance. It is worth noting that the design space size of the NN benchmark is relatively small (48 configurations). For larger design spaces it will be computationally more efficient to use the proposed model rather than checking all possible solutions for different transistor budgets.

Table 3.5: All configurations for region 2 of the NN benchmark

parameter	Conf1	Conf2	Conf3	Conf4	Conf5	Conf6
memory controller	1	1	1	1	2	2
constant cache	512B	512B	1KB	1KB	512B	512B
DL1 cache	8KB	16KB	8KB	16KB	8KB	16KB
register file	8KB	4KB	8KB	4KB	8KB	4KB
IPC	39.32	52.71	39.74	52.59	48.78	59.28
cost	65.36	65.86	66.17	66.67	65.66	66.16
parameter	**Conf7**	**Conf8**	**Conf9**	**Conf10**	**Conf11**	**Conf12**
memory controller	2	3	3	3	4	4
constant cache	1KB	512B	512B	1KB	512B	512B
DL1 cache	8KB	8KB	16KB	8KB	8KB	16KB
register file	8KB	8KB	4KB	8KB	8KB	4KB
IPC	48.90	52.83	61.55	52.49	54.98	**62.38**
cost	66.47	65.96	66.46	66.77	66.26	66.76

3.3.1 Results

Table 3.6 shows the Plackett-Burman results for the benchmarks. Each column marked with the name of the parameter reports the effect S_α of the said parame-

Table 3.6: Resulting parameter effects after Plackett-Burman for all benchmarks

Bench.	MCB	Rank	DL1 cache	Rank	const cache	Rank	RF	Rank
AES	57100	1	17742	4	35500	2	18922	3
FWT	127799	1	73903	2	—	-	459	3
LIB	1482938	3	2284182	2	1051960	4	6037368	1
LPS	340704	1	136082	2	—	-	16736	3
Montcarlo	445073	1	293537	2	46469	3	26229	4
NN	1054553	2	1508787	1	5307	3	535	4
Scan	12372	1	9708	2	—	-	1748	3
Srad	139159	1	2125	2	—	-	117	3
Blackschole	3110257	1	142613	2	—	-	29369	3
Hotspot	433926	1	133392	2	—	-	4584	3
Ray	34441615	2	38417897	1	1405	4	5684461	3
Matrix	13693	1	7931	2	—	-	1775	3
Backprop	14864	1	12020	2	—	-	476	3

ter. The columns marked by "Rank" represent the rank of the effect of the parameter to its left. Thus, for the NN benchmark, we see that the DL1 Cache is ranked as first, that is this benchmark's performance is influenced primarily by the size of its DL1 Cache. It is interesting to note that for most of the benchmarks, the number of memory controllers influences the performance the most, indicating that the higher memory bandwidth may improve performance significantly, seconded by the size of the DL1 cache, while the sizes of the Constant Cache and the Register File have a lesser impact. However, for some benchmarks, e.g., NN, Ray, LIB and AES, the parameters' rank follows a different order. For example, AES is composed of a single kernel which has 257 blocks of 8 warps. Two blocks can run concurrently on each shader which increases the significance of the number of memory controllers to provide sufficient bandwidth. As AES is optimized to store constants in constant memory the performance is sensitive to the size of constant memory. On the other hand, few local memory accesses reduce the sensitivity of performance to the DL1 cache. LIB runs two kernels, each having 64 blocks of 64 threads. Excessive register file usage of each thread limits the number of concurrent blocks. Similar explanations on the parameter sensitivity can be obtained for the other benchmarks.

Figure 3.3 shows the performance (as IPC) of the best (gray bars) and model-suggested (black bars) configurations for all regions of each benchmark. On average, there are 6 configurations in each region. As it can be seen, the optimization algorithm suggested solutions match the optimum performance ones as found through exhaustive search of all possible configurations, except for nine cases. Namely the solutions suggested by the optimization procedure for Regions 1 and 4 of the AES and Regions 4 and 5 of the Srad and Regions 2, 3, 4, 6 and 8 of the Ray benchmarks,

have performance that is slightly lower than the optimal solution found through exhaustive search. For these cases, the performance of the configuration obtained by the optimization method is the second best solution among 14 configurations per region, averaged over all the benchmarks.

The average error in predicted performance is about 0.8% with a maximum error of 3.5%. We compared the results generated by the proposed model against randomly generated results. The random model, randomly selects one of the possible configurations from each region as the optimum configuration. We repeat the random process for one million iterations and report the average error. On average, the performance error of the randomly selected configuration relative to the actual optimum configuration is about 35%. On average, 16% of the times the random model generated results better than the proposed model. As the results indicate the proposed model is more informed than the random model.

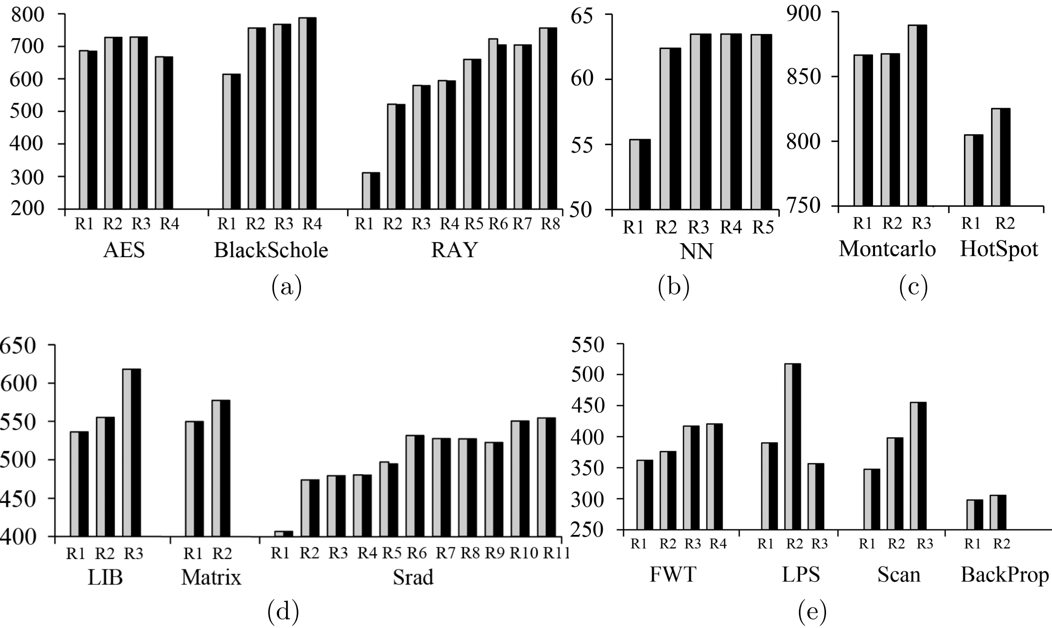


Figure 3.3: Performance Comparison between the Model-suggested and the best performing configurations for different benchmarks

3.4 Related Work

Jia et al. [42] proposed an automated GPU performance exploration framework, named Stargazer, based on stepwise regression modeling. Stargazer simulates de-

sign points which have been sampled randomly from a full GPU design space and uses these sample simulation to build a performance estimator. The results showed that using 0.03% of full design space, Stargazer predicts the performance of any design point with less than 1.1% error.

Palermo et al. [72] proposed a DSE methodology for application-specific multiprocessor system-on-chip. The proposed methodology uses design of experiment techniques, e.g. random and full factorial, to find a set of good candidate architecture configurations to minimize the number of simulations. Then using response surface modeling and the simulation results, an analytical representation of the system's objective function is generated.

Couvreur et al. [97] applied Multi-dimensional Multiple-choice Knapsack Problem (MMKP) to design a MultiProcessor System-on-Chip (MPSoC) runtime manager. Whenever the environment is changing, the runtime manager considers the specification of available application mappings, information provided by the design time exploration, the platform information, and the user requirements to select operating points that minimize total energy consumption. Lilja et al. [96] applied Plackett-Burman design to identify key processor parameters and analyzed their effect on processor enhancement.

As far as we know, at the time of conducting this research, our model was the first to linked the performance of the GPU to architectural parameters under transistor budget limitation. The model development phase requires a few simulations to calculate the influence of different confrontational parameters on the performance and can easily be extended to larger problems with more parameters.

3.5 Conclusion

In this chapter, we presented a method that establishes a GPU configuration that adheres to a transistor budget limit and obtains the optimum or near optimum performance. Among all parameters in GPU design space, we used four parameters, i.e. register file size, shared memory and data cache sizes and number of memory controller, that we could estimate the costs in term of transistor count. We did not include the remaining of the parameters, because of the difficulty in calculating the costs. Therefore, the solutions the proposed model yields are sub-optimal solutions as it does not consider the complete solution space.

Our method delivered the optimum performing configuration in 48 out of 57 cases,

and for the nine cases where the configuration did not achieve optimum performance, its performance lagged the optimum one by around 3.5% maximum error and 0.8% average error. Our method is very efficient in that it requires far fewer simulations to achieve its results. For example, for the case of the SRAD benchmark, an exhaustive design exploration would have required the simulation of 144 configurations, while our method arrives at the optimum configuration by simulating just 16.

The method we presented in this chapter was built based on the assumption that the changes in the configuration of the GPU linearly affects the performance. In the next chapter we will show that the effect is not linear and propose a model that learns the non-linear relationship between the architectural parameters and the power and performance of a GPU.

Chapter 4

Neural Network Based GPGPU Power and Performance Predictor

Application and architectural parameters influence the power and performance of GPUs. Studying all possible configurations of an application on a real GPU is not feasible because of I) the time consuming nature of the problem and II) limitations in configuring architectural parameters. GPGPU power and performance simulators and analytical models can be used for this purpose and they provide more flexibility to configure architectural parameters. The downside of using simulators is that the execution time is longer than running the application on a real hardware.

In the previous chapter we employed Plackett-Burman design and linear integer programming to predict optimum configurations for GPGPU applications under area constraints. That model showed that the performance can be modeled as a function of the architectural parameters of the GPU . A limitation of that model was the assumption that the performance is linearly related to the parameters. However, as we showed in Figure 2.1 of Chapter 2, the power and performance relations almost linear with the architectural parameters except for the extreme values of the parameter. Therefore, the linear relationship assumption reduces the search space of the solution, leaving out some of the possible optimum solutions. In order to capture this non-linearity, in this chapter, we propose to model the power and performance as a function of architectural parameters of the GPU. The proposed model uses a neural network to learn this relationship and predict the power and performance of GPGPU applications based on the GPU configuration.

The model is composed of three steps.

- In the first step, the model reduces the size of the design space by limiting the range of each configuration parameter to values that are actually contributing to performance. In this step, the model eliminates GPU configurations that have low hardware resource utilization which means they are power inefficient and are not of interest of the system designer.
- In the second step, the model utilizes a neural-network based predictor and a small randomly selected subset of the design space for training proposes to predict power and performance of all configurations in the design space.
- In the third step, the model employs a heuristic to evaluate the accuracy of the predictions made by the model and indicate the possible outliers which will be excluded from the final results.

The neural network-based prediction model that we use was introduced to address the problem of generalization. It is an ensemble neural network system and has been used previously in Quantitative Structure–Activity Relationship (QSAR) studies in particular to predict enzyme-inhibitory activities [34]. The predictor uses mechanisms and heuristics to train one or more neural networks with high generalization abilities.

In our case, we train the predictor based on examples of GPU architectural configurations and the performance and power of specific benchmarks running on those configurations. To each of the configurations there corresponds an exemplar comprising a vector of descriptors and the expected (experimentally derived from simulations) power and performance. Since the power/performance characteristics are also benchmark specific, for this work, we develop benchmark specific predictors.

It is worth mentioning that the GPU configuration is determined statically and the configuration will be fixed for the entire execution of any benchmark that runs on the GPU. Also, we assume the GPU is dedicated for computing and not graphic processing.

The configuration descriptor comprises (a) the number of memory controllers, (b) register file size, (c) shared memory size, (d) constant cache size, (e) data-level-one cache size, (f) data-level-two cache size, (g) instruction cache size and (h) the amount of parallelism. We have also introduced the effect of each parameter on the power and performance to the predictor by adding Plackett-Burman Design [76] results to the descriptor vectors.

The predictors accurately predict power and performance of GPGPU applications for most of the configurations in the design space with average prediction error of less

than 6.5%. It is worth mentioning that the error of the NN-based model is larger than the model we presented in the previous chapter, because the domain of the previous model was sub-domain of the NN-based model for which we consider more configurational parameters. Also, in the previous model we employed the Plackett-Burman design technique which explores only a range of each configurational parameter that maintains the linear relationship between the performance and the parameters. But, the NN-based model goes beyond that and explores a wider range for each parameter. Therefore, the NN-based model searches a much larger solution space as it is not limited by the assumption that the power and performance values are linearly related to the configurational parameters of the GPU. For configurations with high prediction errors, an outlier detection mechanism is used to filter them out from the output of the predictor. The proposed filter captures most of the extreme outliers and improves the accuracy of the predictor.

We have evaluated the proposed model using four GPGPU benchmarks, i.e., AES, CP, LPS and STO [8], with various design space sizes [1]. The range of the architectural parameters are listed in Table 4.1. In order to determine the accuracy of the predictor, we simulated the whole design space for each benchmark and compared it against the predictor's outputs. The trained predictor could successfully predict the activity of the remaining configurations in the design space with mean error as low as 3.5% (AES), 1.77% (CP), 6.5% (LPS) and 6.5% (STO).

Although the mean error of the predictor is low, there are some configurations with large prediction errors which we call outliers. The prediction errors of the outliers go as high as 69%, 54% 102% and 89% for AES, CP, LPS and STO, respectively. Although outliers comprise a small portion of the design space, it is desirable to be able to determine and exclude them from the output of the predictor. To this end, we have devised methods to detect and filter out outliers using the characteristics and geometry of the design space as well as the process of developing the NN predictor. We proposed this outlier detection method in our previous work [49]. Using the proposed outlier detection mechanism, we could filter as many as 35%, 9%, 27% and 7% of the outliers for AES, CP, LPS and STO, respectively.

As we shall see in Chapter 5 we have improved the computational complexity of the training method and were able to model 16 more benchmarks.

The chapter is organized as follows. Section 4.1 represents background on GPU

[1]The GPU clock frequncy is the same for all the configurations in the design space. We do not consider the effect of changing the configurational parameters on the clock frequency of the GPU.

architecture, simulations, benchmarks and the NN model used. Section 4.2 discusses our methodology. Section 4.3 presents the results and Section 4.4 concludes.

4.1 Background

4.1.1 GPU Architecture

We evaluate the proposed method on NVIDIA's Fermi GPU architecture. We presented the detailes of the Fermi in Chapter 1 on page 11. Table 2.2 on page 36 summarizes the configuration of the GPU that is simulated in this work. However, the values of the configuration parameters, i.e. the number of memory controller blocks, the size of register, shared memory, DL1, Data Level 2 (DL2) and Instruction Level 1 (IL1), vary for each configuration in the design space (c.f. Table 4.1).

4.1.2 Simulations and Benchmarks

We have used GPGPU-Sim version 3.1.2., a cycle-level GPU power and performance simulator [5]. For each example in the training set we have configured the simulator and obtained the average power and performance metrics. We have evaluated the proposed model with four benchmarks[2], i.e., STO with 420 design space configurations, LPS with 1280 configurations, CP with 3432 and AES with 23040 configurations. It takes around 60, 600, 1400 and 2000 hours to simulate the STO, LPS, Cp and AES benchmarks, respectively. A short description of each benchmark is presented in Table 2.1 on page 35. We used a randomly selected, small subset of the design space for training purposes, 5%, 5%, 5% and 2.5% of the design spaces of the STO, LPS, CP and AES benchmarks, respectively.

4.1.3 Descriptor Vector Expansion

The descriptor vector for each GPU configuration includes 7 parameters as mentioned earlier in this chapter. Through experimentation we found that including information on the sensitivity of the performance as calculated by the Plackett-Burman approach,

[2]To evaluate the accuracy of the model, we simulate the whole design space for each benchmark which, depending on the size of the design space, can take days to complete. Therefore, we could not afford to evaluate the model with all the benchmarks we had access to.

improved the accuracy of the resulting model. We have therefore expanded the descriptor vector by including an additional descriptor, named PB, the value of which is derived through the Pluckett-Burman approach.

As discussed in Chapter 3, Section 3.1.1, we obtain the magnitude of the effect of each parameter to identify the importance of the parameter in determining performance (or power). After obtaining the effects of all parameters for a configuration C_i, we calculate a single value for the configuration by adding up all the effects weighted by the value of their corresponding parameter. Each of the weights is normalized to the minimum value of the corresponding parameter. The calculated value shows the total influence of the architectural parameters of the GPU on the power or performance. We denote the number of units of a parameter α by W_α and calculate the Plackett-Burman value for configuration i as follows:

$$PB_i = \sum_{\alpha=A}^{G} W_{\alpha,i} * S_{\alpha,i} \tag{4.1}$$

where $W_{\alpha,i}$ is the normalized value (weight) of parameter α for configuration C_i and $S_{\alpha,i}$ is the Placket-Burman effect of parameter α on performance (or power). As an example of determining the normalized weights consider that parameter A is the size of the register file. In the design space under consideration, its minimum value is 2KB. Assume that configuration C_i was a register file of size 3KB. Then $W_{A,i} = \frac{3KB}{2KB} = 1.5$

Plackett-burman design considers only the minimum and maximum of each parameter. If there is a non-linear relationship between the parameters and the power or performance, the Plackett-Burman design may not capture this non-linearity. We propose to explore the influence of the configurations in three regions of the design space separately, i.e., the whole design space, lower sub-space of the design space and upper sub-space of the design space.

To this end, we use the median of each configuration parameter to identify the upper and Lower design sub-spaces. The upper sub-space comprises parameters whose values vary between their corresponding median and maximum while for the lower subspace the parameter values vary between their median and minimum. Any configuration may belong either to the upper or lower subspace, or it may not belong to either of these two subspaces. For each of the subspaces plus the whole design space, we apply the Placket-Burman approach separately, and calculate three sets of S parameters. Then for each configuration, we determine whether it belongs to either of the two subspaces when upon, we calculate the PB descriptor using the S

parameters corresponding to the identified subspace. If the configuration does not belong to either of the two subspaces, then we use the S parameters corresponding to the whole design space.

Alo, we learned through our experiments that the balance between the number of memory controller blocks, DL1 and DL2 cache sizes plays a significant role in the system's response. Therefore, we introduced an additional descriptor which represents how well these parameters are balanced. For each configuration we calculate this as follows:

$$NewDescriptor = \frac{MCBs * DL2}{DL1} \tag{4.2}$$

in which MCB is the number of memory controller blocks.

4.1.4 NN Ensemble

In this section we discuss the methodology we use to create the model. The particular method that we have used is an ensemble method.

In a supervised learning technique, the model uses a sample data set with known outputs to relate the outputs of the system as a function of the inputs. Then, the function is used to predict the output of the system for new data. How well the model generalizes its learning to new data is critical. In a NN model, some of the initial parameters of the model, i.e. learning rate, weights, training and validation sets, are chosen randomly. Every time that the model is trained, it ends up with a slightly different mapping function, and therefore, it may end up in a different, perhaps, local minimum[3].

The process of training NNs is an ill-posed problem [59], in that infinitely many solutions satisfy the training criteria, i.e. of having minimum error for the examples in the training set. Generalization, that is choosing the correct function that will respond correctly to an unknown example requires some kind of regularization, i.e. imposing an additional criterion that will limit the choice of the infinite different solutions of the training process. The NN ensemble we use trains many models and produces the final result as a combination of the outputs of the trained models [31, 51]. The model has been used successfully on a number of applications including fault classification [74] and to predict biological activity in QSAR problems [34].

[3]The loss function of a neural network is usually not convex and may have many local minimums.

The universe of discourse for the present study, comprises the sets of configurations of a GPU. Each set of configurations (customized for each benchmark) is divided into a blind set and a model development set (Figure 4.1). The blind set is used to test the accuracy of the model, and does not participate in its development. The model development set is a small subset of the design space comprising between 2.5% to 5% of the design space.

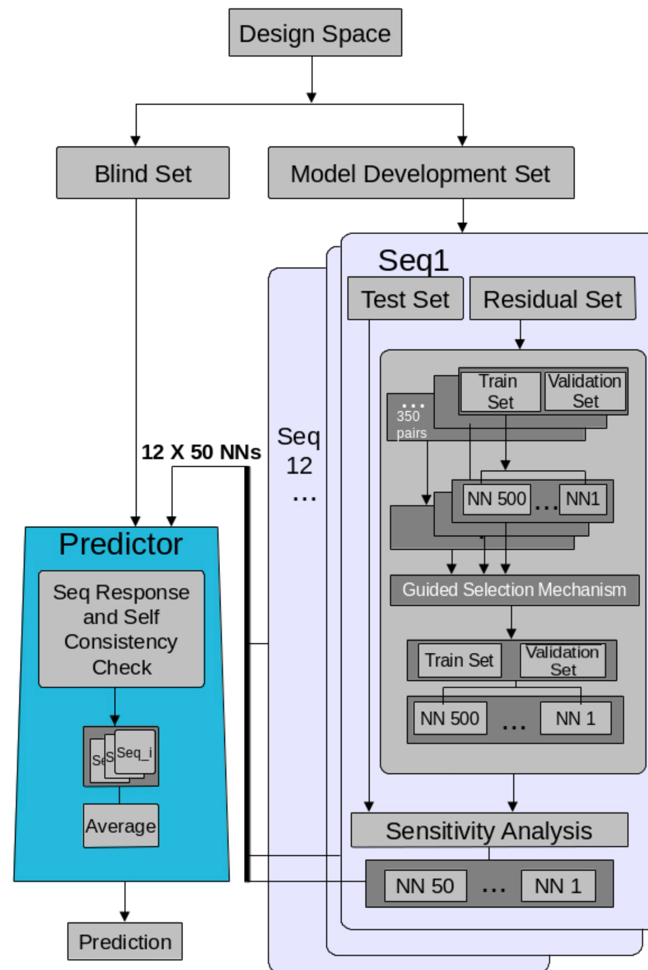


Figure 4.1: The block diagram of the NN ensemble.

To provide the model multiple views of the problem, our methodology creates 12 randomly selected test sets from the model development set and the balance of configurations is allocated to the corresponding residual set. Each pair of the test and its corresponding residual sets is called a sequence. The test set is used to choose the models with the best generalization capability and its size is application dependent and is determined based on the size of the data set and complexity of the problem. We

use test sets of 3, 4, 6 and 20 configurations for STO, LPS, CP and AES benchmarks, respectively.

The residual set is used to construct models. For each sequence, the residual set is further divided into the validation set and the training set. The training set is used to train the NNs and the validation set is used to prevent over-training. The general approach is to select the validation set randomly. However, if the configurations in the validation set are quite different from the configurations in the training set, the NN cannot make an accurate predict for it as it has not seen such configuration in the training phase. On the other hand, if we choose the validation set from configurations that their information is carried by the remaining configurations in the training set, we provide the model with more information potentially improving its generalization abilities.

To divide the residual set into training and validation sets, we use the *Guided-selection* [51] heuristic that identifies the exemplars that are unique and dissimilar from the ones in the training set and excludes them from the validation set. Using the leave-one-out method, the Guided-selection heuristic determines how unique are the various exemplars by attempting to train a NN without the said exemplar and calculating the NN response for the left out exemplar to determin the generalization capability of the resulting model. It identifies the exemplars that, when left out, resulted in the highest error from the aforementioned group of NNs, as the exemplars with "high information content" and selects the validation set from the rest of the exemplars.

Now that the exemplars in the training and validation sets are identified, for each sequence, we use these sets to train another large group (500) of NNs. We use these trained neural networks in the next steps of development of our model.

In our ensemble model the assumption is that the domain of discourse is smooth. To expose the smoothness characteristics of the domain of discourse, the Sensitivity heuristic exposes the set of trained neural networks obtained earlier to perturbed exemplars. We expect that minor perturbations of the input to a set of trained NN would not have a significant effect on their responses. Therefore, the Sensitivity heuristic discards the NNs that their responses change significantly. The remaining neural networks will have a similar behavior towards the effects of the noise perturbations on their training set elements.

After this step we need to integrate the responses of these sequences and make predictions for the configurations in the blind set. For each exemplar in the blind

set, we construct a different specialized model comprising NNs from a subset of the sequences. We choose the sequences which, first, their NNs respond in a similar manner to the said exemplar or their responses to be statistically close to each other, i.e. follow a narrow Gaussian distribution. Second, we want the response distribution of different sequences to be similar to each other. We use statistical tests e.g. t-test to identify sequences that are not self similar. We reject these and retain the remaining ones. The average response of the remaining sequences to the blind exemplar is deemed the response of the model. This is the model refinement heuristic. For the complete description of this procedure, please refer to [49].

4.2 Methodology

As we mentioned at the beginning of this chapter, to make predictions for the configurations in the blind set, the model goes through three steps; a) determining the effective design space, b) exploring the design space and c) detecting the outliers. In this section we present details of each step.

4.2.1 Determining The Effective Design Space

The first thing that the proposed model does, is to decide a range for each configuration parameter. It does that, by limiting the range of each parameter to the values that contribute to GPU performance. For example, the number of memory controller blocks determine the memory bandwidth and the more of these units we have on the chip, the higher the memory bandwidth will be. On the other hand, the performance saturates after a certain number of this unit which is application specific.

In this step, for each benchmark, we perform an exhaustive test for each configuration parameter to determine its effective range. To this end, for a given parameter, we set all other parameters to their extreme values, to make sure that they are not performance bottleneck, and then we vary the parameter of interest until the performance saturates.

For example, to determine the range for the DL1 cache for AES benchmark, we set all other parameters to extreme values, e.g. number of MCBs to 15, DL1 cache size to 4 MB etc. Then we simulate the benchmark for configurations that their DL1 cache varies from 500-Byte to 1 MB (16 different configurations). Then we limit the range of the DL1 cache for the AES benchmark to the values that influenced

the performance. On average, for each benchmark we need around 70 simulations to determine the range of all 7 parameters. At the end of this step, for each benchmark, the size of the design space is determined and the model moves on to the next step.

Table 4.1 reports the configuration parameters that are obtained from the first step of the model and are used to create the design space as well as their size variations for each benchmark.

Table 4.1: Range of variable GPU configuration parameters.

Parameter	LPS	STO	AES	CP
MCB	2-5	3-5	1-6	1-6
Shared memory	4-32 KB	32-64 KB	8-64 KB	48 KB
DL1 cache	1-8 KB	1-8 KB	1-3 Kb	4-32 KB
DL2 cache	1-8 KB	1-4 KB	500 Byte-4 KB	12-32 KB
Inst. cache	2 KB	8-32 KB	3-6 KB	500 Byte-1 KB
Register file	4-32 KB	16-32 KB	4-32 KB	2-16 KB
Constant cache	8 KB	8 KB	500 Byte-4 KB	500 Byte-6 KB

4.2.2 Constructing the Model

To explore the design space, the model uses a NN-based predictor, as discussed in Section 4.1.4 to accurately predict power and performance of the GPU configurations. The predictor uses a heuristic method to filter the outliers.

Normalization and Scaling

To each of the configurations there corresponds an exemplar comprising a vector of descriptors defining the GPGPU configuration and the expected (experimentally derived from simulation) power and performance. Before training the predictor, we process and normalize the data set. We use Principal component analysis (PCA) and we normalize the data using Standard Scores (Z-score [29]) to convert all the descriptors to a common scale with mean of zero and standard deviation of one while keeping the shape of the distribution of the descriptors. After Z-score normalization, we divide the values of each descriptor by the maximum absolute value of the normalized descriptor to scale the data between -1 and 1 [4].

[4]For AES, we didn't use PCA and Z-score and only normalized the data by scaling them between -1 and 1.

NN Predictor Development

We presented a detailed discussion about how our NN model are developed in Section 4.1.4. For each benchmark, we use only a small percentage (5%) of the exemplars to train our ensembles. The trained ensembles, on average make very accurate predictions, but the models are not error-free and there are outliers. In the next section we present our outlier detection heuristic.

4.2.3 Outlier Detection

The model we discussed in the previous section makes accurate predictions for most of the configurations in the Blind set. However, it miss-predicts some of the exemplars. We define a prediction as an outlier if the prediction error is larger than the mean of errors plus two standard deviations. This is quite possible, since the blind set may include exemplars that represent functionality that is completely different from the one expressed by the exemplars included in the training set.

In order to attempt to identify such possible outliers, we have developed a filtering heuristic that attempts to identify such aberrant behaving exemplars. The rationale of the heuristic developed is centered on the role the test set plays in the training of the NN. The test set is used to identify the NN that would generalize correctly. Therefore, for an unknown configuration, we examine how far or close its response is from the responses of the test set configurations that are close to it. If the configuration is not an outlier we should see small difference. Otherwise the difference should be large meaning that the NN fails to generalize correctly. We proceed as follows.

As discussed in Section 4.1.4, at this stage, for each unknown exemplar (blind), the model has identified a number of h sequences that comprise the blind-exemplar specific model. We extract the exemplars that comprise the test sets of the said sequences and select k closest neighbors to the unknown exemplar[5]. The idea for choosing only the k closest neighbors from the test set is that configurations that fall close to each other in the space of configuration parameters, should have similar responses (i.e. power or performance). We have examined different numbers for k and the results we have obtained show that choosing 4 closest neighbors produces the best results for most of the benchmarks. Then, we obtain the difference between the average response of the test set exemplars (R) and the response of the unknown exemplar (R') as follows:

[5]We use the Euclidean distance to calculate the distance between two configurations.

$$Diff = R' - \frac{1}{k*h}\sum_{i=1}^{h}\sum_{j=1}^{k} R_{i,j} \quad (4.3)$$

which i and j enumerate the sequences and the closest neighbor exemplars in each sequence, respectively.

The filter identifies 5% of the configurations in the Blind set with the largest response difference as possible outlier. Our experiments showed that 5% of the blind set configurations with highest distance values includes most of the extreme outliers with minimum false positives. The filter is implemented in Matlab and the source code can be found in Appendix A.2.

4.3 Results

In this section we report power and performance prediction and the outlier detection heuristic results evaluated on NVIDIA's GTX480 Fermi GPU architecture (Table 2.2 on page 36) using GPGPU-Sim [5]. We have evaluated the model with four benchmarks as the process of simulating all design space configurations and training the NN models are time consuming mainly due to the number of NNs that the *Guided-selection* heuristic trains to identify the validation set. In Chapter 5 we propose a fast and accurate alternative to the *Guided-selection* heuristic that enables us to evaluate more benchmarks.

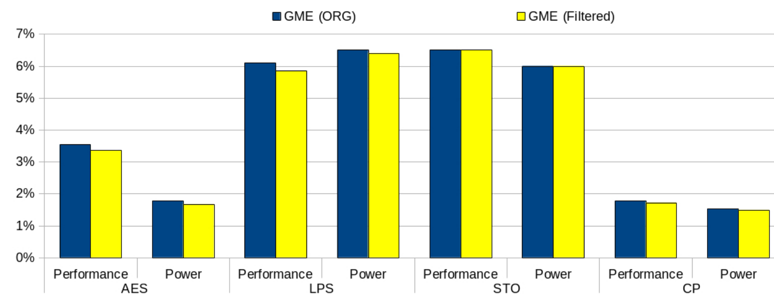
In order to determine the accuracy of the predictor, we simulated the whole design space of the benchmarks and compared them against the predictions generated by the predictor. We train the predictor, as outlined in Sections 4.1.4 and 4.2.2 and predict the power and performance of the the remaining of the design space. Outliers are detected through filtering as per section and 4.2.3.

Table 4.2 and Figure 4.2 summarize the results of the power and performance predictors for the AES, CP, LPS and STO benchmarks (labeled original). As it can be seen from the table, despite the low average error there exist outliers which comprises less than 5% of the validation sets over all predictors for three benchmarks (column marked as "% of outliers (original)" in Table 4.2).

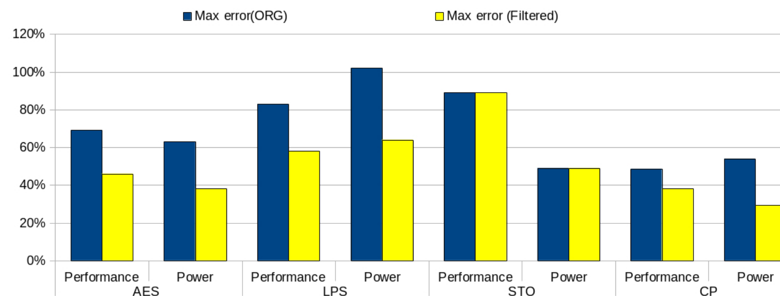
In Table 4.2 we report the performance of the predictor that utilizes the filter (labeled filtered). As the table shows, for AES and LPS, the filter captures the extreme outliers for both power and performance predictors and improves the geometric mean of errors (GMEs) of both predictors. For CP, the filter improves the GME but fails

Table 4.2: The power and performance prediction results.

Benchmark	Predictor	GME (original)	Max error (original)	% of outliers (original)	GME (Filtered)	Max error (Filtered)
AES	Performance	3.54%	69%	5%	3.36%	46%
	Power	1.77%	63%	4%	1.67%	38%
CP	Performance	1.77%	48.6%	3.8%	1.72%	38%
	Power	1.53%	53.94%	3.90%	1.49%	29.37%
LPS	Performance	6.1%	83%	5%	5.85%	58%
	Power	6.5%	102%	5%	6.4%	64%
STO	Performance	6.5%	89%	3.5%	6.5%	89%
	Power	6%	49%	5%	6%	49%



(a)



(b)

Figure 4.2: (a) GME and (b) maximum error before and after applying the filter.

to capture the extreme outliers. For STO, the filter does not improve the GME of the predictors and fails to capture any of the extreme outliers. We think that this is due to the fact that the number of configurations in the filter is quite small (21 configurations for STO).

Table 4.3 reports the filter coverage, which is the percentage of the outliers that is captured by the filter, and the efficiency of the filter, which is the percentage of the

Table 4.3: The filter coverage and efficiency.

Benchmark	predictor	Filter coverage	Filter efficiency	remaining outliers
AES	Performance	28.8%	35%	3.6%
902 out of 22540	Power	35%	28%	2.7%
CP	Performance	8.94%	6.75%	3.62%
	Power	3.91%	3.07%	3.97%
LPS	Performance	27%	25%	3.9
	Power	8.34%	8%	4.8%
STO	Performance	7%	5%	3.4%
	Power	0%	0%	5.3%

Table 4.4: The coverage and efficiency of the random filter.

Benchmark	predictor	Filter coverage mean, best	Filter efficiency mean, best	remaining outliers mean, best
AES	Performance	5%, 4.9%	5%, 6.5%	5%, 4.9%
	Power	5%, 6.6%	4%, 5.3%	4.1%, 4%
CP	Performance	5.03%, 9.76%	3.79%, 7.36%	3.77%, 3.58%
	Power	5.03%, 8.59%	3.94%, 3.75%	3.92%, 3.78%
LPS	Performance	5.4%, 16.6%	5.1%, 15.7%	4.9%, 4.3%
	Power	5.2%, 18.7%	4.9%, 17.6%	4.9%, 4.2%
STO	Performance	5.4%, 28.5%	3.6%, 19%	3.4%, 2.6%
	Power	4.9%, 23.8%	4.9%, 23.8%	5.1%, 4.1%

configurations in the filtered data that are actually outliers. The table also report the percentage of the outliers in the remaining data after filtering. As it can be seen, the proposed filter performs well and reduces the percentage of the outliers in the remaining data for most of the cases.

We also compare the performance of the filter to a filter that randomly selects the same number of configurations. We run the random filter for 100 times, and compare the average and the best of those 100 runs to the result obtained from the proposed filter. Table 4.4 reports the random filter results. As the table shows, the proposed filter has better coverage and efficiency and leaves less outliers in the remaining data than the random filter (except for the power predictors of the CP and STO benchmarks). Also, it outperforms the best results obtained through random filter for both models of the AES and the performance model of the LPS benchmark.

In Figure 4.3 we compare the efficiency of the filter before and after applying the filter to the data. As can be seen for most of cases, except the power model of the

STO benchmark, the filter reduces the percentage of the outliers in the data.

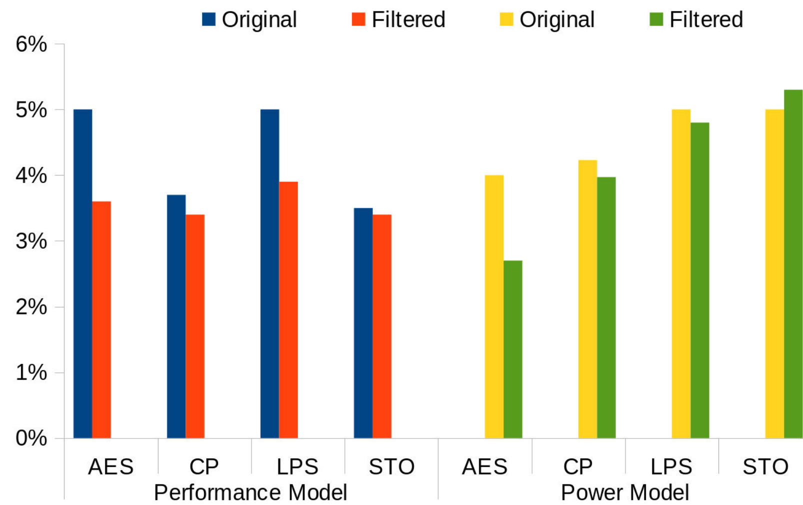


Figure 4.3: Filter efficiency.

4.4 Conclusion

We proposed a a NN-based application specific power and performance predictor for GPGPU applications. The model uses a small portion of the design space for training and could accurately predict the power and performance of the rest of the design space. The predictor uses an outlier detection heuristic to detect and filter the outliers to increase the accuracy of the predictor. Using the proposed model, one can apply optimization techniques to obtain a subset of the design space that meets certain constrain on power or performance or both. As we shall see in the next chapter, we apply a multi-objective optimization technique to obtain power-performance optimum configurations.

Chapter 5

Accelerating Neural Network Training with Quantum Annealing

One of the factors that we have shown affects the generalization capability of a NN model is the selection of the validation set. In the standard approach, the data set that is used to develop the model is divided into training, test, and validation sets randomly. However, if the validation set includes exemplars that are quite dissimilar to those in the training set, the model would not have adequate information to reach a correct decision. The NN-based GPGPU power and performance predictor introduced in Chapter 4 utilizes the Guided-selection heuristic [51] to select the most-appropriate members of the training and validation sets.

Although the Guided-selection heuristic resulted in the construction of a robust model, however, it increases the time required to train the model. On average, the Guided-selection heuristic takes about 90% of the training time. In this chapter, we propose an alternative approach to Guided-selection heuristic that can be formulated as quadratic optimization problem. We use a variety of efficient solvers including quantum annealing infrastructure to solve it. The results show radical improvement in the development time of the model, while preserving accuracy. This technique is used as the alternative to the Guided-selection heuristic in the model presented in Chapter 4 to select the validation and training sets. This new model is evaluated with the same set of benchmarks used in Chapter 4 for comparison. After we show that the new model that utilizes this technique have the same generalization ability as the original model, then we evaluate it with more benchmarks. The detailed discussion on the ensemble NN and GPU architecture and benchmarks can be found in the model

presented in Chapter 4.

Our contribution is as follows:

1. We propose alternative approaches for our Guide-selection heuristics by formulating it as an optimization problem.

2. We propose the use of mathematical solvers and quantum annealing techniques to solve the optimization problem.

3. We show that the alternative approach accelerate the NN model training while preserving accuracy.

4. We evaluate the alternative approach with 16 more benchmarks and report the NN-predictor results and the performance of the outlier filter.

The rest of this chapter is organized as follows. Section 5.1 goes over methods used to solve optimization problem. Section 5.2 discusses our methodology. Section 5.3 presents the results, and Section 5.4 concludes.

5.1 Optimization Methods

5.1.1 Exhaustive Search

Given a cost function $(C_{\bar{\sigma}})$, where $\bar{\sigma}$ is a vector of parameters, the unconstrained optimization problem is the one that determines a particular vector $\bar{\sigma}_{opt}$ that optimizes the cost function, that is, $\bar{\sigma}_{opt} = \text{argmin}_{\bar{\sigma}}(C_{\bar{\sigma}})$. In performing an *exhaustive search*, one evaluates the cost function on all possible values of the parameter vector and selects the value that minimizes the cost function. Such an approach is exceedingly computationally expensive. Therefore, computationally efficient heuristic approaches have been developed that do not guarantee the optimal solution, but yielded an acceptable sub-optimal one.

5.1.2 Tabu Search

Tabu search is a well-known metaheuristic search algorithm which tries to avoid getting trapped in local minima by remembering the recently visited best configurations and disallowing revisiting such configurations for a certain number of iterations [25], [26], [78].

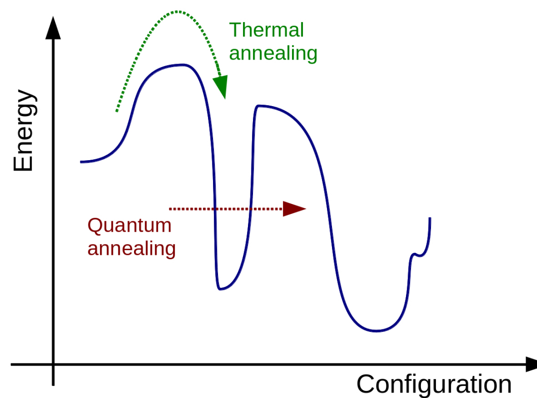


Figure 5.1: Quantum annealing vs thermal annealing.

5.1.3 Quantum Annealing

Quantum annealing utilizes the well-known quantum tunneling effect to ensure that a system can escape local minima as it explores the state space of an energy function on its way to settling in a ground state. Quantum annealing relies on quantum fluctuations, and by gradually reducing the magnitude of fluctuation anneals the system down to its minimum cost state. The quantum fluctuations help tunneling through the energy barriers and avoid local minima. In thermal annealing, however, the system climbs the individual energy barriers by utilizing thermal fluctuations. This is shown in Figure 5.1. Therefore, quantum annealing is more effective in solving multi-variable optimization problems [10].

D-Wave Systems has implemented a sequence of quantum computational systems that use quantum annealing to find the ground state of an artificial Ising system [43]. The latest implementation of these systems include up to 2000 superconducting flux quantum bits (qubits) with programmable spin–spin couplings. The behavior of such a system is described by an Ising Hamiltonian as

$$H_p = \sum_{i=1}^{N} h_i \sigma_i^z + \sum_{i,j=1}^{N} J_{ij} \sigma_i^z \sigma_j^z \tag{5.1}$$

where σ_i^z is the z-component of the Pauli matrix for spin i, h_i is the energy bias, J_{ij} is the coupling energy between spins i and j, and N is the number of qubits in the system. Quantum annealing on such a system is performed by slowly evolving the system Hamiltonian [43]

$$H(t) = \Gamma(t)\sum_{i=1}^{N}\Delta_i\sigma_i^x + \Lambda(t)H_p, \tag{5.2}$$

where Γ decreases from 1 to 0 while Λ increases from 0 to 1 as time passes, σ_i^x is x-component of the Pauli matrix for spin i, and Δ_i parameterizes quantum mechanical tunneling between two lowest states of the system. If the annealing is done sufficiently slow, the system remains at the found state, ending up at the ground state of H_p at the end of the annealing. The Hamiltonian of Equation 5.1 can be written in vector form as $H(s) = s^TJs+s^Th$, expressing a quadratic unconstrained binary optimization (QUBO) problem [99].

Many problems in machine learning, pattern recognition, and deep neural networks can be formulated as combinational optimization problems to take advantage of quantum annealing. Such problems are notoriously difficult to solve, as in Equation 5.2, $H(t)$ depends on σ and the complexity of finding the combinations of σ that optimize $H(t)$ is exponential to the number of σ. The advantage of quantum annealing is that it converges to an optimum solution faster than other techniques.

5.1.4 SQA

Simulated Quantum Annealing (SQA) [16] is a numerical method that is used to emulate quantum annealing using Quantum Monte Calro (QMC) numerical simulations [83]. SQA can be used in simulating time-dependent quantum systems.

5.2 Methodology

An alternative approach to the Guided-selection heuristic is to select the validation set from the exemplars that are most similar to exemplars in the training set. We propose, therefore, to partition the Model Development Set into a training and validation set so as the sum of the distances of the elements of the validation set from the training set would be minimum. This can be expressed as a constraint optimization problem as follows:

$$\begin{aligned} \sigma = \text{argmin}_{\sigma_i} \quad & \sum\nolimits_{i=1}^{N}(1-\sigma_i)\sum\nolimits_{j}(\sigma_j \Delta_{ij}) \\ \text{provided that} \quad & T(\sigma) = \left(\sum_{i=1}^{N}\sigma_i - M\right)^2 = 0 \end{aligned} \tag{5.3}$$

where σ_i is one if i is in the training set and zero if it belongs to validation set, N is the total number of exemplars in the model training set, and M is the size of the training set. Assuming an exemplar is denoted by a vector of d descriptors, e.g. $m = [m_1, ...m_d]$, Δ_{mn} is the distance between two exemplars m and n, and is calculated as follows:

$$\Delta_{mn} = \sqrt{\sum\nolimits_{k=1}^{d}(m_k - n_k)^2} \tag{5.4}$$

Additionally, we need to ensure that the number of data chosen is exactly M, thus, selecting σ so that $T(\sigma) = 0$ expresses this constraint:

$$\begin{aligned} T(\sigma) = \left(\sum_{i=1}^{N}\sigma_i - M\right)^2 &= \sum_{i,j}\sigma_i\sigma_j + M^2 - 2M\sum_{i=1}^{N}\sigma_i \\ = \sum_{i,j}\sigma_i\sigma_j + M^2 - 2M^2 &= \sum_{i,j}\sigma_i\sigma_j - M^2 \end{aligned} \tag{5.5}$$

Adding a constraint term to the cost function ensures that the number of data chosen for the training set is exactly M. The cost function is expressed as follows:

$$\sigma = \text{argmin}_{\sigma_i} \quad \sum\nolimits_{i=1}^{N}(1-\sigma_i)\sum\nolimits_{j}(\sigma_i \Delta_{ij}) + \sum\nolimits_{i,j}\sigma_i\sigma_j - M^2 \tag{5.6}$$

5.3 Experimental Setup and Results

This section presents the experimental results of applying heuristic techniques, including quantum annealing, to the optimization problem described in Sections 5.2.

In the models we have developed in chapter 4, we used the ensemble NN model as a power-performance predictor to explore the design space of GPGPU applications. To evaluate the accuracy of the model, we simulated all of the configurations with

GPGPU-SIM [5]. Table 5.1 shows the benchmarks used to evaluate the method, their design space sizes, their training and validation set sizes, and the search space that the optimization technique must explore. The training and validation sets are determined using the Guided-selection heuristic.

Table 5.1: GPGPU benchmarks

Benchmark	Design Space Size	Training Set Size	Validation Set Size	Number of Solutions
AES	23040	480	4	2,184,297,480
CP	3432	168	4	32,018,910
LPS	1024	48	4	194,580
STO	432	19	3	969

5.3.1 Guided-selection Heuristic Alternative Results

The optimization problem is solved using exhaustive search, the tabu solver, SQA, and quantum annealing. For quantum annealing, we used the D-Wave 2X [22] quantum computer from D-Wave. It implements a quantum annealing algorithm using 1000 qubits. It solves a Quadratic Unconstrained Binary Optimization (QUBO) problems. Binary variables and correlations between them are mapped to qubits and couplings between them, respectively [22]. As the size of problems suitable for solving using the D-Wave 2X computer is limited[1], we were able to solve the optimization problem for only the STO benchmark. For comparison, we produce the solutions obtained by the Guided-selection heuristic as well as by Tabu and SQA for all four benchmarks. Note that the Guided-selection heuristic does not partition the model development set based on the minimum distance of exemplars in the training and validation sets. Rather it accomplishes the partitioning based on the actual generalization abilities of trained NN on the respective training and validation sets. We have postulated that the partition based on the minimum distance between the training and validation sets will result in generalizing capable NN. This section will examine whether is hypothesis is correct. To this end we will do two comparisons. First whether the QUBO formulation of the problem and its solutions are equivalent to the exhaustive search and second, whether the minimum distance partition will produce similar quality generalizing NN.

[1] The qubits are not fully connected which is required to have a controllable connection between qubits to act collectively.

Table 5.2 reports the distance values and the execution time for each solver while section 5.3.2 presents the accuracy of the model when the alternative heuristic is employed.

Table 5.2: Distances and timings of different solvers

	Benchmark	Exhaustive Search	Guided Selection	Tabu	SQA	D-Wave 2X
Distance	AES	4454.96	5522	4554.96	4454.96	–
	CP	645.90	820.09	645.90	–	–
	LPS	156.11	183.50	156.11	156.24	–
	STO	87.99	96.70	87.99	87.99	88.15
Timing (seconds)	AES	500,000	180,000	0.15	1324	–
	CP	3,158,000	126,000	0.009	–	–
	LPS	264	72,000	0.001	566	–
	STO	0.6	25,200	0.00001	1.3	70

As Table 5.2 shows, the tabu solver produces the same minimum distance configurations as the exhaustive search does, and with an execution time of less than one second. When we were conducting this study, we did not have SQA solver results for the CP benchmark, but for the other benchmarks, the SQA solver produces optimum results (AES and STO) or results very close to the optimum ones (LPS with less than 0.1% error). Compared to the optimum results, results produced by the D-Wave 2X have less than 0.2% error, on average. It is worth noting that the error made by the Guided-selection heuristic is due to the fact that it does not search for minimum distance configurations; it trains many NNs to select the training set exemplars. However, the results are reasonably close to the optimum and this supports our postulate.

Table 5.2 also shows the execution time of different solvers. The time it takes for the exhaustive search heuristic to produce the optimum distance configurations grows exponentially with the size of the problem. For example, we ran the exhaustive search for AES on four nodes, each equipped with an Intel Xeon Phi processor and 115 GB of RAM, and, utilizing 240 cores in total, it took more than six days to calculate the optimum results. For the other three benchmarks, the exhaustive search was run on a single core. As the size of the problem grows, the Guided-selection heuristic becomes faster than the exhaustive search heuristic. The tabu solver produces optimum results in less than one second. The SQA solver is slower than the tabu solver, as it simulates the quantum annealing process. The run time of the D-Wave 2X system is greater than expected, as it requires tuning, pre-processing, and post-processing to get the

optimum or close to optimum results.

5.3.2 Incorporating the Proposed Mechanism into the Ensemble Neural Network Model

In this section, we study the effect of using our alternative method for Guided-selection heuristic in a NN-based performance predictor and compare the results with the results produced by the original method. Table 5.3 reports the geometric mean of errors, the maximum error, and the number of outliers (that is, where prediction errors are larger than 15%) of the NN-based predictors using validation sets produced by different solvers. Although the exhaustive search, Tabu and SQA have identical solutions (table 5.2), NN model results are slightly different for different solvers (see Table 5.3) because for each run of the NN model, initial weight values are set randomly.

Table 5.3: NN-based predictor performance comparison for different methods

	Benchmark	Exhaustive Search	Guided Selection	Tabu	SQA	D-Wave 2X
Geometric Mean of Errors	AES	2.46%	3.54%	2.33%	2.33%	–
	CP	1.80%	1.78%	1.78%	–	–
	LPS	6.12%	6.1%	5.92%	5.98%	–
	STO	8.62%	6.5%	8.81%	8.35%	8.90%
Maximum Error	AES	351%	69%	405%	297%	–
	CP	46%	48%	39%	–	–
	LPS	106%	83%	89%	95%	–
	STO	108%	89%	139%	102%	138%
Number of Outliers	AES	367	1132	296	321	–
	CP	117	123	139	–	–
	LPS	42	48	48	50	–
	STO	22	14	18	22	25

In general, the results of using the validation set obtained through the alternative optimization path are close to the ones obtained using the original Guided-selection heuristic. It seems that in the case of AES, where the search space is very large, the solution obtained using optimization is better than the one obtained using the Guided-selection heuristic. Also, the number of outliers is smaller in the optimization-obtained validation set. However, the opposite is true when the search space is considerably smaller, as in the case of STO.

It seems that when the number of exemplars is small (STO), the more informed Guided-selection heuristic is more effective, whereas when the number of exemplars is large, searching a validation-guided selection that is very similar in structure to the training set improves the results.

5.3.3 Evaluating the Model with More Benchmarks

We showed that choosing the validation and training sets using the optimization technique instead of the Guided-selection heuristic does not affect the generalization capability of the ensemble NN. The new approach significantly reduced the time it takes to select the validation set which is approximately 90% of the training time. For example, as Table 5.2 shows, for AES, the new approach is 1.2 million times faster than the Guided selection heuristic. This enables us to evaluate the model with more benchmarks.

we can now evaluate the model with more benchmarks.

In this section we evaluate the model with 16 more benchmarks and present the performance of the ensemble NN model and the outlier filtering heuristic (See Chapter 4 for details). The optimization technique is solved using SQA (see section 5.1.4 for details).

Table 5.4 reports the benchmarks and the range of architectural parameters.

Table 5.5 summarizes the results of the power and performance predictors (labeled original). As it can be seen from the table, despite the low average error there exist outliers which comprises less than 6% of the blind sets over all predictors for all benchmarks (column marked as "% of outliers (original)" in Table 5.5).

We pass the prediction results through our outlier detector to exclude the outliers from the results. In Table 5.5 we report the performance of the predictor that utilizes the filter (labeled filtered). The filter tags 5% of the output of the predictor as potential outliers. As the table shows, for most of the benchmarks, the filter captures the extreme outliers for both power and performance predictors and improves the Geometric Mean of Error (GME)s of both predictors. For Gaussian and the performance model of Transpose, the filter improves the GME but fails to capture the extreme outliers. For the power model of Transpose, the filter does not improve the GME and fails to capture any of the extreme outliers. We think that this is due to the fact that the number of configurations in the filter is small (5 and 11 configurations for Gaussian and Transpose, respectively).

Table 5.4: Range of variable GPU configuration parameters.

Benchmark	MCB	Register file (KB)	Shared memory (KB)	Constant cache (KB)	DL1 (KB)	DL2 (KB)	IL1 (KB)
NN	1-8	4-16	8	8	1-16	1-32	2
RAY	1-5	8-32	8	8	1-32	1-8	4-8
Needle	1-7	2-8	8	8	1-32	1-256	2
Hotspot	1-3	16-32	4-16	8	2-32	0.5-16	1-2
Backprop	1-7	8-32	2-8	8	0.5-256	0.5-32	0.5-1
SRAD	1-7	8-32	8-32	8	0.5-32	0.5-128	1-4
Gaussian	1-3	4	8	8	0.5-8	0.5-4	0.5-1
DWT	1-7	8-32	8	8	0.5-8	0.5-32	0.5-1
Transpose	1-8	8	8	8	8	0.5-256	0.5-2
EigenValues	6-8	16-64	16-64	8	8	4-16	0.5-4
ConvTexture	1-8	8-64	8	8	8	0.5-64	0.5-2
MergeSort	1-5	16-64	8-32	8	0.5-64	0.5-64	0.5-2
ConvSeparable	1-7	4-32	8-32	8	0.5-16	0.5-64	2-8
MUM	1-7	8-16	8	8	0.5-64	0.5-256	1-4
Histogram	1-3	4-32	8-32	8	0.5-64	0.5-32	0.5-2
WP	1-5	4-32	8	8	1-16	1-8	1-16

Table 5.6 reports the filter coverage, which is the percentage of the outliers that is captured by the filter, and the efficiency of the filter, which is the percentage of the configurations in the filtered data that are actually outliers. The table also reports the percentage of the outliers in the remaining data. We also compare these results to a filter that randomly selects the same number of configurations and report the results in the table (in parenthesis we report the average and the maximum values of 100 runs of the filter).

As it can be seen in the table, the proposed filter performs well and, on average, it has better coverage and efficiency and leaves less outliers in the remaining data than the random filter. Also, it outperforms the best results obtained through random filter (maximum values) for both performance and power predictors.

5.4 Conclusion

In this chapter we studied the performance of quantum annealing techniques when used in training ensemble Neural Networks.

We identified a sections of the training process, i.e. Guided-selection of exemplars where the heuristics involved could be expressed as quadratic unconstrained optimization problem.

We evaluated the proposed approach both on the quality of the solutions obtained by a variety of solvers including the D-Wave 2X system, and the impact the proposed approach has on the training of our ensemble NN.

Our study shows that quantum annealing techniques (i.e., SQA and D-Wave 2X) produce optimum or near optimum solutions to the optimization problem studied. Further, our study showed that by replacing the original heuristic of our training methodology with the proposed optimization method produced models that in most cases perform better than the models produced by our original method.

Table 5.5: The power and performance prediction results.

Benchmark	Predictor	GME (original)	Max error (original)	% of outliers (original)	GME (Filtered)	Max error (Filtered)
NN	Prf	4.11%	88%	3.74%	3.78%	53%
	Power	3.14%	63%	4.68%	2.91%	53%
RAY	Prf	8.21%	379%	2.5%	7.36%	137%
	Power	5.98%	75%	4.31%	5.65%	70%
Needle	Prf	7.93%	106%	5.53%	7.35%	82%
	Power	6.45%	81%	5.2%	5.99%	68%
Hotspot	Prf	6.35%	113%	3.08%	6.03%	91%
	Power	5.83%	89%	2.89%	5.57%	69%
Backprop	Prf	7.02%	157%	3.31%	6.48%	115%
	Power	4.6%	63%	4.2%	4.29%	60%
SRAD	Prf	8.95%	177%	3.21%	7.87%	137%
	Power	6.91%	81%	4.31%	6.48%	75%
Gaussian	Prf	7.16%	46%	5%	6.69%	46%
	Power	3.14%	28%	5%	2.95%	28%
DWT	Prf	7.39%	130%	2.99%	6.74%	104%
	Power	5.99%	106%	2.92%	5.54%	73%
Transpose	Prf	2.14%	25%	2.92%	2.04%	25%
	Power	0.13%	5%	2.5%	0.14%	5%
EigenValues	Prf	0.72%	4.5%	3.7%	0.68%	3.6%
	Power	0.12%	0.6%	2.77%	0.12%	0.5%
ConvTexture	Prf	7.21%	141%	3.78%	6.68%	89%
	Power	6.5%	73%	4.82%	6.01%	54%
MergeSort	Prf	4.5%	206%	3.43%	4.23%	54%
	Power	2.64%	37%	5.07%	2.46%	31%
ConvSeparable	Prf	7.69%	175%	4.2%	6.96%	103%
	Power	5.66%	70%	4.06%	5.32%	66%
MUM	Prf	8.81%	414%	3.54%	7.54%	104%
	Power	6.23%	72%	4.14%	5.87%	67%
Histogram	Prf	8.94%	166%	3.82%	8.35%	105%
	Power	7.65%	79%	4.44%	7.21%	73%
WP	Prf	5.18%	69%	4.21%	4.84%	52%
	Power	4.62%	73%	4%	4.32%	63%

Table 5.6: The filter coverage and efficiency.

Benchmark	predictor	Filter coverage (Rand mean, Rand Max)	Filter efficiency (Rand mean, Rand Max)	remaining outliers (Rand mean, Rand Max)
NN	Prf	50% (4.9%, 16.7%)	40% (4%, 13.8%)	1.96% (4.2%, 3.6%)
	Power	33% (5%, 17.9%)	33% (5.1%, 17.2%)	3.27% (5.2%, 4.6%)
RAY	Prf	77% (5.7%, 22.2%)	41% (3.2%, 4.4%)	0.58% (2.8%, 2.27%)
	Power	48% (4.6%, 13%)	44% (12.5%, 14.8%)	2.33% (4.8%, 4.4%)
Needle	Prf	43% (4.2%, 14.3%)	51% (5.1%, 17.1%)	3.31% (6.1%, 5.5%)
	Power	29% (4.6%, 13.2%)	31% (5%, 14.3%)	3.88% (5.5%, 5%)
HotSpot	Prf	47% (5.9%, 16.4%)	29% (4.7%, 11.9%)	1.73% (3.9%, 3.3%)
	Power	38% (5.4%, 18%)	22% (4.6%, 13.6%)	1.88% (4.2%, 4%)
BackProp	Prf	47% (5.2%, 9%)	33% (3.1%, 5.2%)	1.84% (3%, 2.9%)
	Power	41% (5.2%, 8.6%)	36% (3.9%, 6.5%)	2.62% (3.8%, 3.7%)
SRAD	Prf	49% (5%, 7.5%)	33% (3.4%, 5.1%)	1.72% (3.4%, 3.3%)
	Power	39% (5%, 7%)	35% (4.5%, 6.4%)	2.75% (4.5%, 4.4%)
Gaussian	Prf	50% (6.5%, 40%)	60% (6.4%, 40%)	2.61% (4.5%, 2.9%)
	Power	33% (3.7%, 33.3%)	40% (4.4%, 40%)	3.48% (5.6%, 3.9%)
DWT	Prf	57% (5.1%, 16%)	36% (3.4%, 10.6%)	1.36% (3.3%, 2.9%)
	Power	51% (5.3%, 21%)	33% (3.4%, 13.6%)	1.5% (3.2%, 2.7%)
Transpose	Prf	43% (4.5%, 28.6%)	27% (2.8%, 18.2%)	1.75% (3.1%, 2.3%)
	Power	50% (4.7%, 33.3%)	27% (2.6%, 18.2%)	1.31% (2.6%, 1.8%)
EigenValuse	Prf	50% (5.4%, 25%)	40% (4.3%, 20%)	1.94% (3.9%, 3.1%)
	Power	33% (4.3%, 22.2%)	20% (2.6%, 13.3%)	1.94% (2.9%, 2.4%)
ConvTexture	Prf	41% (5%, 17.3%)	34% (4.1%, 14.3%)	2.32% (4.2%, 3.7%)
	Power	38% (4.8%, 13.5%)	40% (5.1%, 14.7%)	3.14% (5.4%, 4.9%)
MergeSort	Prf	39% (5.2%, 11%)	28% (3.7%, 7.9%)	2.19% (3.6%, 3.4%)
	Power	40% (5.2%, 8.9%)	42% (5.6%, 9.5%)	3.21% (5.3%, 5.1%)
ConvSeparable	Prf	57% (5%, 7%)	50% (4.4%, 6.2%)	1.90% (4.4%, 4.3%)
	Power	38% (4.9%, 7.5%)	32% (4.2%, 6.4%)	2.66% (4.3%, 4.2%)
MUM	Prf	61% (4.7%, 9.2%)	46% (3.5%, 6.9%)	1.44% (3.7%, 3.5%)
	Power	42% (5.2%, 10.1%)	36% (4.5%, 8.8%)	2.53% (4.3%, 4.1%)
Histogram	Prf	39% (4.7%, 10%)	32% (4%, 8.5%)	2.43% (4.3%, 4%)
	Power	35% (4.9%, 11.3%)	33% (4.8%, 11.1%)	3.0% (4.9%, 4.6%)
WP	Prf	31% (4.7%, 11%)	28% (4.2%, 9.8%)	3.02% (4.4%, 4.1%)
	Power	36% (4.7%, 10.1%)	30% (3.9%, 8.5%)	2.66% (4.2%, 4%)

Chapter 6

MultiObjective GPU Design Space Exploration Optimization

The model we discussed in the previous chapter predicts the power and performance of the configurations in the design space of a GPGPU benchmark. The filtering heuristic is used to detect and exclude the configurations that are likely to be outliers from the output of the model. As the size of the design space could be as large as thousands of configurations, to efficiently search for a configuration that meets a certain design goal, in this chapter we propose to use multi-objective Pareto Front (PF) optimization technique. Therefore, in this chapter we further develop the model presented in previous chapter and apply Pareto Optimal technique to the results generated by the predictor to obtain a subset of the design space of a GPGPU application that its configurations are power-performance optimum. After obtaining application-specific optimum configurations, we propose a method that searches the Pareto Optimal configuration space of all benchmarks and finds a configuration that performs well over any set of applications.

The rest of this chapter is organized as follows. In Section 6.1 we review the Pareto Optimal optimization technique. In Section 6.2 we present the methodology and simulations and benchmarks we have used to evaluate our technique. Section 6.3 presents the results and Section 6.4 concludes.

6.1 Pareto Optimal Optimization Technique

Having the responses (i.e. power and performance) of all the configurations in the design space, one can seek configurations that meet certain restrictions on power and performance such as configurations that offer high performance with minimum power dissipation. Such configurations can be obtained using the Pareto Front optimization algorithm [17].

The problem as stated is one of a multi-objective optimization. Since in a multi-dimensional space, one cannot establish an ordering relation, one rather uses the dominance relation.

Each configuration is associated with a certain power and performance. These are obtainable either directly through simulation of a benchmark, or indirectly predicted by a model. To describe the Pareto Front algorithm, we associate to each configuration a characteristic. A characteristic of configuration i is the vector $X_i = [f(x_i), g(x_i)]$ where f and g are real functions that associate power and performance values to x_i, respectively. Although the characteristic is defined specifically as a two-dimensional vector of functions of power and performance, in general, its dimensionality may be larger than 2 and comprise functions of arbitrary cost characteristics.

In general, given a set of m characteristics $S = (X1, \dots X_m)$ and n objectives $(f_1(S), ...f_n(S))$, a characteristic X_1 is said to dominate a characteristic X_2 (also written as $X_1 \succ X_2$) iff [102]

$$\forall i \in \{1, \dots, n\} : f_i(X_1) \geq f_i(X_2) \quad \wedge \quad \ni j \in \{1, \dots, n\} : f_j(X_1) > f_j(X_2) \qquad (6.1)$$

Once the dominance relation is defined, one can define the non-dominated set of characteristics S'. Given a set of characteristics S, the non-dominated set S' is a subset of S such that each element in S' is not dominated by any other element not belonging in S'.

If the set of configurations associated with the characteristics S encompasses the design space, then the non-dominated set S' is called the globally Pareto-optimal set or simply Pareto Optimal set. To obtain the Pareto-optimal set (a.k.a Pareto Front) one systematically adds configurations to the Pareto-optimal set if they are non-dominated by any of the configurations already in the Pareto Front. The first

characteristic added is one that is optimum with reference to one of the objectives[1]. Details can be found in [63]. The Pareto Front optimization code implemented in Matlab is presented in Appendix A.3.

6.2 Methodology

In the NN-based predictor we developed in the previous chapter, we used the ensemble NN to predict the power and performance of all GPU configurations in the design space for each benchmark and applied our filtering technique to the predicted values to detect and exclude the possible outliers. Now, we apply Pareto Front optimization techniques to obtain the power-performance optimum configurations.

The Pareto Front optimization algorithm requires that all the objectives are optimized identically, i.e. all of them maximized or minimized. Since power and performance are opposing, i.e. one seeks high performance and low power, we have negated the performance so as our new objective is that of minimum power and minimum negative performance.

6.2.1 Scaling the Power and Performance values

The performance and power values of the GPU architectures have different ranges (this can be seen in Figure 2.1 on page 32). As we will be using the Euclidean distance between two configurations in power-performance space to calculate the error of the model, we need to scale both performance and power values of the benchmarks to a fixed range to ensure that both power and performance are weighted equally and to make the calculated distances and therefore the errors comparable across all the benchmarks. Therefore, before performing any calculation, we scale both power and performance to the same range of 1 to 10 using the following equation.

$$prf_i' = \frac{(prf_i - prf_{Min}) * (10-1)}{prf_{Max} - prf_{Min}} + 1 \tag{6.2}$$

where prf_i is the performance (or power) of the configuration we want to scale, prf_i' is the scaled performance (or power), prf_{min} and prf_{max} are the minimum and maximum performance (or power) of the original data.

[1]Descriptions for Pareto Optimal algorithm in this section are paraphrased from [63] and the terminology adapted to refer to the problem at hand, i.e. that of seeking configurations that exhibit maximum performance at low power.

6.2.2 Quality of Obtained Predicted Pareto Front

Since the performance and power values that the model predicts are not without prediction errors, we want to determine how close to the Actual Pareto Front is the Pareto Front obtained through the developed predictor (Predicted Pareto Front). As the size of the Predicted Pareto Front set is usually small, we simulate the Pareto Front configurations and obtain the actual power and performance values to compare with the Actual Pareto Front configurations. Thus from this point forward, we shall use the developed power and performance models to choose the Pareto Front configurations, however, the values of the power and performance corresponding to the said configurations will be obtained from actual, i.e. simulations. Thus the term Predicted Pareto Front will refer to the set of configurations obtained by using the developed power and performance models, however, the power/performance values corresponding to these configurations will be the actual ones obtained through simulation.

We shall define some notation and terminology following which, we shall present our method.

Denote by $\boldsymbol{F}^l$, the Pareto Front associated with code B^l. Then the Pareto Front $\boldsymbol{F}^l$ is a set of configurations C_i^l each running the same code B^l i.e. $\boldsymbol{F}^l = \{C_i^l; i = 1, \ldots K\}$. We shall use the notation C_j to simply denote a configuration without consideration of which code it runs, while the notation C_j^l denotes configuration j running code B^l. With each configuration C_i running code B^l, we associate the vector $\mathbf{P}_i^l = (pwr_i^l, prf_i^l)$ of the average power and performance achieved. We also associate the characteristic vector $\mathbf{X}_i^l = (pwr_i^l, -prf_i^l)$. The characteristic vector is used in the Pareto Front algorithm and ensures that both power and performance are optimized identically i.e. both power and the negative of performance are both *maximized* or *minimized* as required by the Pareto algorithm.

Denote the Actual Pareto Front by $\boldsymbol{F}^l$ and the Predicted Pareto Front by $\boldsymbol{F}'^l$, given a configuration C'^l_j from $\boldsymbol{F}'^l$, we denote its distance δ_j^l from the Actual Pareto Front $\boldsymbol{F}^l$. The distance is defined with respect to the Actual Pareto Configuration that is closest to C'^l_j. The closest configuration to C'^l_j is denoted as $\widehat{C'}^l_j = C^l_{\underset{i}{\text{argmin}}(d_{ij})}$ and $d_{ij} = \|\mathbf{P'}^l_j, \mathbf{P}_i^l\|$. i.e. $C_i^l \in \boldsymbol{F}^l$.

The distance δ'^l_j can now be defined as the normalized difference of the power-performance pair of the Predicted Pareto Front configuration C'^l_j from its closest Actual Pareto Front configuration $\widehat{C'}^l_j$ as defined above. That is $\delta'^l_j \equiv \widehat{\delta'}^l_j = \frac{\|\mathbf{P}^l_j, \widehat{\mathbf{P}}^l_j\|}{\|\widehat{\mathbf{P}}^l_j\|}$.

We use ${\delta'}^l_j \equiv \widehat{{\delta'}^l_j}$ as the relative error (RE) between two configurations. All the power and performance values are actual values obtained from simulations. After obtaining the relative errors of all configurations in the Predicted Pareto Front to the closest Actual Pareto Front, we calculate the geometric mean of relative errors.

Expanded Predicted Pareto Front:

The cardinality of the Predicted Pareto Front is most often smaller than that of the Actual Pareto Front. We have elected therefore to expand the Predicted Pareto Front by including configurations that are close to the configurations included in the Actual Pareto Front. We proceed as follows.

Denote bye D^l the design space with n configurations associated with code B^l. For each configuration C_j in $\boldsymbol{F'^l}$ we calculate its distance from each configuration in the design space, i.e. ${\delta'}_{j,D}$. Then, we interpolate the distribution of the distance values in ${\delta'}_{j,D}$ with a Gaussian distribution[2] and denote the the standard deviation by ${\sigma'}_j$ and the average of the standard deviations of all configurations in F'^l by $\bar{\sigma}'$. We use 1% of the $\bar{\sigma}'$ as the distance threshold (Dis_Thr) as it resulted to a expanded pareto set equal in size to that of teh actual ones. Then we examine each configuration C_i in D^l to determine if it is within the Dis_Thr distance to any of the configurations in $\boldsymbol{F'^l}$ and create the Expanded Pareto Front set, i.e. $\boldsymbol{EF'}$:

$$
\begin{aligned}
& EF' = F'; \\
& \forall i \in \{1, \ldots n\} \quad and \quad \forall {\delta'}_{i,j} \in \delta_{i,F'} : \quad if \quad {\delta'}_{i,j} \leq Dis_Thr \\
& then \quad EF' = EF' \cup C_i
\end{aligned}
\tag{6.3}
$$

Using the standard deviation as a closeness metric depends on the shape of the design space and may not work for some data sets. For example, for a design space with a small variation in the distance values, (i.e. small standard deviation and Dis_Thr) this method may not expand the Predicted Pareto set no matter how close the configurations are to Pareto Fronts. However, this method worked for all the benchmarks we used in this study and resulted in an Extended Pareto Front that is close in cardinality to the Actual Pareto Front. The results show that the relative errors between Actual Pareto Front and Extended Pareto Front are relatively the same as the relative errors between Actual Pareto Front and Predicted Pareto Front. Therefore we decided no to report those results.

[2] We use Matlab's "*fit*" that fits a curve or surface to data.

6.2.3 Random Sets

To demonstrate that the models we have developed are informed, we compare the Predicted and Expanded Pareto Fronts with identical-cardinality sets that are generated randomly.

Thus, utilizing Matlab's "*randsample*" function, we generate a random set by randomly choosing configurations from the design space, equal in number to the cardinality of the Predicted Pareto Front. Subsequently, we expand this random set using the same process outlined in Section 6.2.2. We repeat this process for 9604 times which provide results with 0.01% confidence interval and 95% confidence level. For each iteration of the process, we obtain the relative errors of configurations in both the random set and the expanded random set to the configurations in the Actual Pareto Front using the procedure outlined in Section 6.2.2 above (Equation 6.3). Then we report the geometric mean of all 9604 relative errors. We also report the percentage of the random samples with relative errors larger than the relative errors of the Predicted Pareto Front and Expanded Pareto Front.

6.3 Results

In Chapter 5, we reported the power and performance predictions made by our NN model for 20 benchmarks. We use the same simulations and prediction results as reported in Section 5.3.2 and Section 5.3.3. In this section we present the results of the Pareto Front optimizer for these benchmarks.

We obtain Pareto Front configurations using power and performance values obtained from simulations (i.e. Actual Pareto Front configurations) and compare them to the Pareto Front configurations predicted by our model (see Section 6.2.2) . We also compare the Predicted Pareto Front configurations with Randomly selected Pareto Front configurations as outlined in Section 6.2.3.

Table 6.1 reports the size of the Actual and Predicted Pareto Front for the benchmarks. Using the actual responses for the Predicted Pareto Front, the average relative error between configurations in the Predicted Pareto Front and the closest configurations in the Actual Pareto Front (marked as "RE (P,A)") are listed in the table. Except for EigenValues, the relative errors are small. The EigenValues only have one Actual Pareto Front configuration that is miss-predicted by the model. As an example, in Figure 6.1, we show the Pareto Front sets for the STO benchmark using

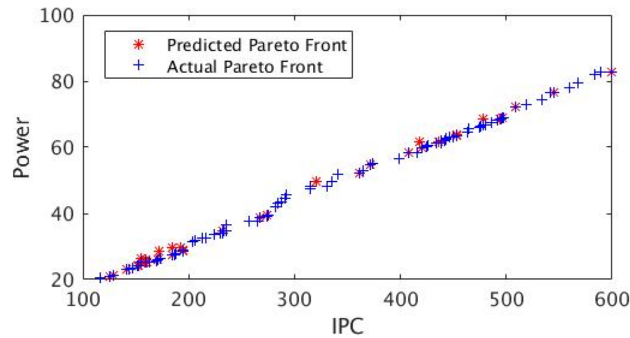


Figure 6.1: Actual and Predicted Pareto Front for STO benchmark.

the simulation results (Actual) and NN predictor results (Predicted). The actual values from simulation results are used for both sets in the graph. As the figure depicts, the Predicted Pareto Front configurations are very close to the Actual Pareto Front configurations. It is worth noting that the number of Predicted Pareto Front configurations are less than the Actual ones.

To compensate for the size difference between Actual and Predicted Pareto sets, we decided to expand the Predicted Pareto Front set to be the same size as the Actual Pareto Front set and examine how it affects the quality of the Predicted Pareto Front set. Table 6.1 reports the relative error between the Expanded Pareto Front and the closest neighbors in the Actual Pareto Front (marked as "RE(E, A)"). For most of the benchmarks, expanding the Predicted Pareto Front result in larger relative errors.

We generate 9604 sets of randomly selected configurations from the design space and generate the random and expanded random sets. We compare these random sets and the expanded random sets to the Actual Pareto Front. Table 6.1 reports the average relative errors of the randomly selected sets to the Actual Pareto Fronts (marked as "GME (R, A)" and "GME (E, A)"). Also, the relative error between expanded random sets and the Actual Pareto Fronts (marked as "GME (ER, A)") are listed in the table. As the table shows, the results generated by our model always outperforms the results that the random process generates.

The last two columns of the Table 6.1 report the fraction of the 9604 random configurations outperforming the Predictor or the Expanded Pareto Fronts in terms of error (labeled RBM and ERBM in the table, respectively). The fraction is reported as a percentage. Note that the size of the Actual and Expanded Pareto fronts are the same. As it can be seen, the model outperforms the Random process for most of the benchmarks, except for EigenValues and Transpose benchmarks (due to their small

Table 6.1: Pareto Front results summary.

	A_{PF} size	P_{PF} size	RE (A,P) $*10^{-2}$	RE (A,EP) $*10^{-2}$	GME (A,R) $*10^{-2}$	GME (A,ER) $*10^{-2}$	RBM	ERBM
NN	95	32	0.07	0.18	0.49	0.59	0	0.03
RAY	87	27	1.34	2.03	2.12	2.53	1.01	6.40
Needle	154	19	0.35	0.31	0.39	0.42	17.8	22.4
Hotspot	48	31	0.67	0.75	1.86	2.04	0	0
Backprop	383	69	0.18	0.19	1.16	1.33	0	0
SRAD	118	141	0.37	0.36	1.29	1.41	0	0
Gaussian	25	16	1.42	1.35	1.97	2.02	2.6	7.3
DWT	327	91	0.09	0.10	0.12	0.12	0.08	2.09
Transpose	13	22	0.16	0.16	0.16	0.16	54	62
EigenValues	1	13	164	176	98	114	85	94
ConvTexture	122	68	0.38	1.62	1.25	1.57	0	5.8
MergeSort	64	51	0.40	0.56	1.82	1.95	0	0
ConvSeparable	137	141	0.54	0.65	1.45	1.68	0	0
MUM	158	88	0.18	0.23	1.26	1.49	0	0
Histogram	117	113	0.32	0.52	2.57	3.72	0	0
WP	113	53	0.56	0.88	1.34	1.35	0	0
LPS	57	41	0.51	0.57	2.07	2.1	0.8	4.8
STO	85	37	0.47	0.49	0.53	0.57	25.9	19.2
CP	304	159	0.23	0.24	0.58	0.59	6.4	3.1
AES	143	105	0.33	0.36	1.12	1.17	0	0

design space size).

6.3.1 Obtaining a Good Configuration for a Set of Application Codes

The Pareto Front sets we obtained in the previous section contain the benchmark-specific power-performance optimum configurations. This means that for a particular benchmark we have identified configurations resulting in optimum, in the Pareto Sense, power and performance. However, since each Pareto Front is associated with a particular benchmark, running an alternate benchmark on one of the Pareto Front

configurations, will result in non-Pareto-Front Power and Performance for the chosen alternate benchmark.

The goal in this section is to develop methods to obtain configurations on which a set of a variety of codes (e.g. benchmarks) would perform "well".

We have purposely used the term "perform well", since there are many alternate interpretations. For our purposes, we understand that a code "performs well" on a configuration, the closer its power-performance metrics are to the Pareto Front of the said code.

In Section 6.2.2 we defined the distance with respect to $\boldsymbol{F}^l$ that is closest to C_j^l. The distance can be defined with respect to the "center" of the Pareto Front or we can use the projections of the distances on the power or performance dimensions as a closeness metric. The center of the Pareto Front is denoted as $\widetilde{C}_j^l$ and it is defined as a fictitious configuration on which the code B^l achieves an average power $\widetilde{pwr}^l = \frac{1}{K}\sum_{i=1}^{K} pwr_i^l$ and average performance $\widetilde{prf}^l = \frac{1}{K}\sum_{i=1}^{K} prf_i^l$.

The distance δ_j^l can now be defined as the normalized difference of the power-performance pair of configuration C_j^l from the center configuration $\widetilde{C}_j^l$ as defined above. That is $\delta_j^l \equiv \widetilde{\delta}_j^l = \frac{\|\mathbf{P}_j^l, \widetilde{\mathbf{P}}_j^l\|}{\|\widetilde{\mathbf{P}}_j^l\|}$.

From the definitions above, one can deduct that $\widehat{C}_j^l$ is zero if the configuration belongs to the Pareto Set. However if one considers the distance to $\widetilde{C}_j^l$, this is not zero, but it is smaller to the center-distance of configurations outside the Pareto Set.

Having defined the notion of the distance δ_j^l of a configuration C_j^l running a particular benchmark code B^l, we need to consider the situation where two or more configurations run 2 or more codes, and develop ways of comparing their efficacy. The notion of the distance of a configuration from a Pareto Set, as defined above, plays a crucial role.

Assume two configurations C_m and C_n each running a number of codes $B^l; l = 1, \ldots N$. For each of the configurations and for each of the codes run, we can derive the distances $\delta_m^l; l = 1, \ldots N$ and $\delta_n^l; l = 1, \ldots N$ from the corresponding Pareto Sets. We can now define the the vector of distances $\Delta_m = [\delta_m^1, \delta_m^2, \ldots \delta_m^N]^T$ associated with configuration C_m and a vector norm $\|\cdot\|$ [3]. We are now in a position to introduce an ordering relation that will compare two configurations running a number of benchmark codes. Essentially, we consider one configuration to be more effective than another configuration when it runs a number of benchmark codes, if the resulting power per-

[3] In this work we consider the l_2 and the l_∞ norms.

formance metrics of the said configuration are "closer" to their corresponding Pareto Sets as compared to the those of the latter configuration. In particular, configuration C_m is more effective than configuration C_n when running a set of Benchmark codes $B^l; l = 1, \ldots N$, denoting it as $C_m \succcurlyeq C_n$ iff the norm of its distance vector is lesser than the corresponding norm of configuration C_n i.e. $C_m \succcurlyeq C_n \iff \|\Delta_m\| \leq \|\Delta_n\|$.

Given now a set of K configurations running any number of benchmark codes, it is easy to determine the configuration(s) C_{opt} that most efficiently run this set of codes. To do this, we utilize the ordering relation $\succcurlyeq$ introduced above as $C_{opt} \succcurlyeq C_m \forall m \in [1 \ldots K]$.

Table 6.2: Overal optimum GPGPU configuration.

	l_2 norm	l_∞ norm	center of PF	performance optimum	power optimum
MCB	1	8	5	7	1
Register file	8	16	32	32	4
Shared	32	64	8	32	8
Constant	8	8	8	8	8
DL1	32	8	8	32	2
DL2	2	8	4	128	0.5
IL1	1	0.5	16	4	0.5
Performance	55.76	154.86	126.26	245.36	40.46
Power	17.39	52.70	42.02	74.28	13.03

Table 6.2 reports the configurations that "perform well" using l_2 norm, l_∞ norm, closest to the center of Pareto Front, performance-only and power-only distance metrics. Also, for each configuration we report the average performance and the average power values over all the benchmarks.

As the table shows, using the proposed method, we could predict the best power, the best performance and a medium power/performance configurations.

6.4 Conclusion

In this chapter we proposed a multiobjective design space exploration optimization model for GPGPU applications. The model uses the GPGPU power performance predictor presented in the previous chapter. We obtained the Actual and Predicted

Pareto Front sets and showed that the Pareto Front produced with our model comprises configurations that are very close to the configurations of the Actual Pareto Front. Such configurations can be used efficiently in power/performance optimization.

The model that we have developed so far is application specific and for every new benchmark we need to build the model and obtain the power and performance values of the configurations in the training set. This can be time consuming as some benchmarks have large design space.

In the next chapter we will propose a new model that relates the power and performance of GPGPU applications to the code attributes.

Chapter 7

Optimum Power-Performance GPU Configuration Prediction based on Code Attributes

In Chapter 4 we developed application specific models that predict the power and performance for a given configuration and with optimization method presented in Chapter 6 we can choose the power-performance optimum Pareto set configurations.

Although the model makes accurate predictions, developing such a model is still time consuming, as a separate model needs to be developed from each application. In this chapter we propose to model GPU architectural parameters as functions of attributes of an application and hence obtain an application independent model. The inputs of the model are GPGPU kernel attributes such as the grid size, thread block size and number of basic blocks. The model predicts a GPU configuration that achieves optimum or near optimum efficiency when the benchmark runs on the predicted configuration. The definition of efficiency depends on design goal. For example in an embedded system one can define it as minimum power dissipation and in a server it can be the maximum performance. We have not dealt with power and defined efficiency as maximum performance. However, the model can easily be modified to yield the minimum power configuration. We refer to the maximum performance configuration of a benchmark as "performance-optimum" configuration.

Figure 7.1 shows the block diagram of the model. The model starts with a set of benchmarks for which their Pareto Front configurations are known. We call these benchmarks the Training Set which is composed of K benchmarks (indicated from

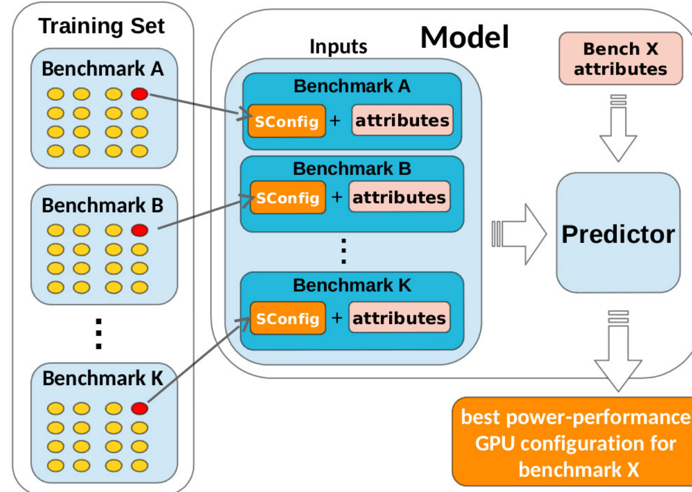


Figure 7.1: Block diagram of the proposed model.

A to K in Figure 7.1). The Pareto Front configurations can be obtained using the optimization method presented in Chapter 6. In Figure 7.1, we show the Pareto front configurations with yellow circles and highlight the performance-optimum configuration with a red circle. A vector of 7 architectural parameters defines each configuration. These are the same parameters used to develop our NN-based model and presented in Section 4.1.1 on page 57.

The range of values of the architectural parameters varies largely from one benchmark to another which means a large configuration space for the model to explore. As The model has only 24 samples (benchmarks) of the space, we have decided to use a coarser granularity by sub-sampling the configuration space by dividing the range of each architectural parameter into a number of sub-ranges. Now, instead of a vector of architectural parameters, we define each configuration with a vector of sub-ranges and call it a "Super-Configuration" (SConfig in Figure 7.1). Therefore, each Super-Configuration represents a subset of the configuration space. This is shown in Figure 7.2. Please note that we are only interested on the configuration space that results to an optimum performance and thus we do not provide the modeling process with the actual power and performance figures. We expect that once a configuration is identified, we can determine the performance and power of a specific application on the identified configuration using simulations or power and performance models. Also, the model receives the attributes of each benchmark which define the benchmark's characteristics.

We have developed two different models, one based on clustering and one based

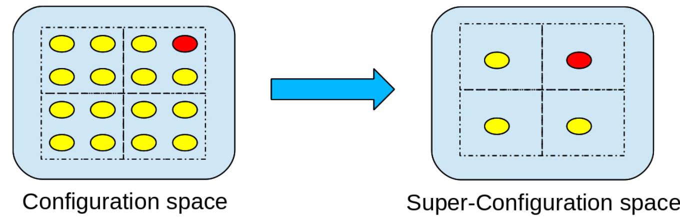


Figure 7.2: Mapping from configuration to Super-Configuration space.

on classification and we compare their results. The models predict the performance-optimum Super-configuration given the code attributes of an unknown code. Given that a Super-Configuration really corresponds to the subranges of configuration parameters, we use exhaustive search (i.e. simulations) of the code on the configurations of the Super-Configuration to identify the performance-optimum one. We have chosen to set the number of clusters and classes equal to the number of sub-ranges. In Figure 7.3, the configurations represented by the Super-Configuration are plotted and the Pareto optimal configurations are marked by green circles. The output of the model is the Pareto optimal configuration that offers the highest performance (the red configuration in Figure 7.3). More details on the clustering and prediction phase of the model will be presented in Sections 7.1 and 7.2.

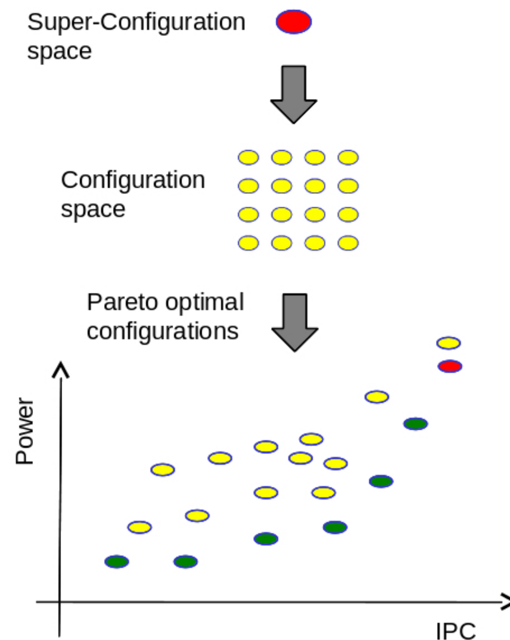


Figure 7.3: Obtaining the power-performance performance-optimum configuration from the predicted Super-Configuration.

To evaluate the quality of the predictions made by the model, we compare the relative error between the predicted configuration for each benchmark and the actual optimum configuration and show that the power and performance of the two are very close. We also compare the result produced by our model with randomly selected configurations and show that the model predicted configuration delivers power and performance closer to actual best configuration compared to randomly selected configuration in the majority of the cases.

In summary our contributions are as follows:

- We study characteristics and attributes of 24 GPGPU benchmarks.
- We propose to model architectural parameters of GPU as a function of the attributes of the application that it runs.
- We propose to use clustering and a NN-based classifier to implement the model.
- We use the proposed model to predict the best power-performance GPU configuration for each benchmark.
- We show that the proposed model can make predictions very close to actual best power-performance configuration.

The rest of this chapter is organized as follows. Section 7.1 represents background on GPU architecture, the profiler we used to study applications' attributes and the clustering and classification methods we used in our model. Section 7.2 discusses our methodology. Section 7.3 presents the results and Section 7.4 concludes.

7.1 Background

7.1.1 GPU Architecture

We evaluate the proposed method on NVIDIA's Fermi GPU architecture as discussed in Chapter 1, Section 1.2 page 11. Table 2.2 on page 36 summarizes the baseline configuration of the GPU that is simulated in this work. The values of some other configuration parameters, i.e. the number of memory controller blocks, the size of register, shared memory, DL1, DL2 and IL1, are determined by the proposed model and are application-specific. Table 7.3 reports the value of these parameters for the power-performance optimum GPU configurations for different benchmarks.

7.1.2 Simulations and Benchmarks

We have used GPGPU-Sim version 3.1.2, a cycle-level GPU power and performance simulator [5]. We have evaluated the proposed model with 24 benchmarks listed in Table 2.1 in Chapter 2. For each benchmark, we evaluate the power and performance of all the configurations in the design space to select the optimum configuration using Pareto Front optimization technique discussed in Chapter 6. We use the Actual Pareto Front configurations in this chapter. However, as we have seen in Chapter 6, the Predicted Pareto Front is very close to the Actual Pareto Front and can be used to save the time required to simulate all design space configurations and the Actual Pareto Fronts.

The range of different architecture parameters can be found in Table 5.4 on page 78. Table 7.2 reports the size of the design space of different benchmarks. As can be seen, we have evaluated benchmarks with different complexities in terms of their memory demands which are reflected in their design space sizes which varies from NQU with 8 to AES with 23040 configurations in their design spaces.

7.1.3 Profiling

To obtain the attributes of GPGPU applications we used NVIDIA assembly code SASS Instrumentor (SASSI) [84]. SASSI is a low level GPU instrumentation tool which allows the user to inject user-written instrumentation code at any instruction specified by the user. SASSI can be used to obtain the application's static and run-time information.

Figure 7.4 shows the block diagram of SASSI instrumentation flow. Application code is compiled and an intermediate code called PTX (parallel thread execution) [69] is generated. SASSI instrumentation codes are embedded into the PTX file. Then, ptxas (PTX assembler) generates binary object files from the PTX file. A backend compiler translates the PTX instructions into machine code to run on the GPU. For the backend compiler, NVIDIA supports ahead-of-time compilation of the kernel code via a PTX assembler (ptxas). To instrument a code, we need to tell SASSI where it should insert instrumentation and what information to extract. For example, to count the number of integer instructions in a GPU application, we can tell SASSI, in the "handler" file (see Figure 7.4), to insert instrumentation code before each integer instruction and inside that instrumentation code, we increment a counter. The instrumented code runs on a GPU and SASSI reports the extracted data in a

formatted file. In this study, we use SASSI to obtain benchmark attributes such as number of kernels, number of basic blocks of each kernel and number of integer instructions of each basic block.

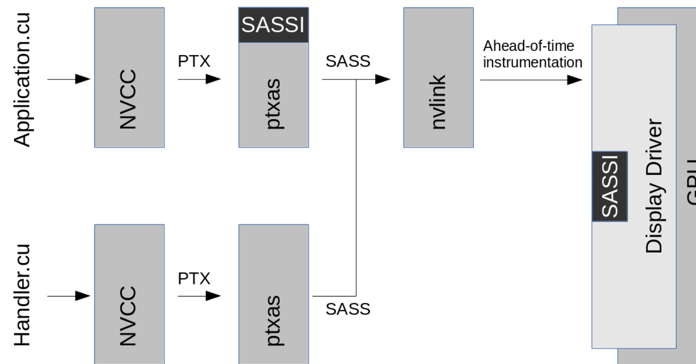


Figure 7.4: Instrumentation flow of SASSI [85].

Table 7.1 summarizes the attributes of the benchmarks obtained using SASSI. As some benchmarks have more than one kernel and each kernel has a different number of threads , we have defined ”dominant parallelism” as an indicator of the amount of parallelism for most of the benchmark's execution time which is the average number of threads running in parallel on the GPU for at least 80% of the benchmark's run time. The number of Integer (Int), floating point (FP), Control Flow (CF), barrier and load/store instructions are averaged per basic block. To reflect the variation of the size of basic blocks and different type of instructions among basic blocks, we also use the average of variance of basic block sizes and each instruction type per basic block in the model (variations are not shown in the table). In total, the attribute vector of each benchmark is composed of 17 descriptors (the average of variances are not shown in the table).

7.1.4 Clustering

Having a set of data points in a N-Dimensional space, the objective of the well known ”K-means” clustering algorithm [56] is to divide the data points into K clusters in a way that the mean of the distances of the data points to their clusters' center is minimum. The algorithm starts by selecting K random points in the N-dimensional space as clusters' centroids. Then, it iteratively assigns each data point to the closest centroid and calculates the mean of the distances and relocate the centroids and updates assignments until the mean of the distances is minimum. As the algorithm

Table 7.1: Applications attributes obtained using SASSI.

benchmark	number of kernels	Grid Size	Block Size	Dominant Parallelism	number of Basic Blocks	Basic Block Size
NN [6]	4	1162	49	120	22	100
LPS [28]	1	100	128	3840	27	5
STO [1]	1	384	128	3840	342	10
CP [86]	1	256	128	1920	19	4
AES [58]	1	257	256	11520	13	22
RAY [60]	1	512	128	9600	87	8
BFS [8]	2	256	256	11520	10	7
NQU [75]	1	256	96	4320	27	5
Needle [8]	255	64	16	240	58	15
HotSpot [8]	1	1849	256	11520	29	6
BackProp [8]	2	4096	256	19200	17	12
SRAD [8]	4	64	256	23040	58	4
VecAdd [64]	1	196	256	7680	4	5
Gaussian [8]	47	9	256	3840	11	8
DWT [64]	2	256	512	7680	13	6
Transpose [64]	24	4	16	720	57	5
EigenValues [64]	5	1	512	7680	71	5
ConvolutionTexture [64]	4	176	192	5760	8	29
MergeSort [64]	17	32	241	10845	64	10
ConvolutionSeparable [64]	6	96	96	4320	10	67
LIB [86]	2	64	64	1920	96	7
MUM [8]	1	20	256	3840	31	6
ScalarProd [64]	1	128	256	3840	23	4
Histogram [64]	12	221	228	10260	79	9

benchmark	number of Integer Instructions	number of Floating Point Instructions	number of Control Flow Instructions	number of Barrier Instructions	number of Load/Store Instructions
NN	47	19	1	0	30
LPS	3	0	1	0	1
STO	6	0	1	0	1
CP	1	1	1	0	0
AES	10	0	0	2	8
RAY	1	4	1	0	1
BFS	3	0	1	0	1
NQU	2	0	1	0	2
Needle	8	0	0	1	6
HotSpot	2	1	1	0	0
BackProp	5	1	0	1	2
SRAD	3	1	1	0	1
VecAdd	2	0	1	0	1
Gaussian	4	0	0	0	2
DWT	3	1	0	0	1
Transpose	3	0	1	0	1
EigenValues	1	1	1	0	1
ConvolutionTexture	3	9	1	4	5
MergeSort	6	0	1	0	2
ConvolutionSeparable	8	27	0	0	29
LIB	2	2	1	0	1
MUM	3	0	1	0	0
ScalarProd	2	0	1	0	0
Histogram	4	0	1	0	3

starts by a set of random centroids the solution may be a local minimum. Repeating the algorithm for several times can increase the chance of finding the global minimum or a minimum close to it.

In this study, to each benchmark we assign a set of code attributes and a set of architectural parameters that describe the optimum configuration. For each architectural parameter, p_i, we cluster the space of the code attributes into a number of clusters, $CL = [cl_1, \dots cl_q]$ in which q is the number of clusters (we try 2 and 3 clusters in this study). Each cluster, cl_j, is assigned with the value, v_j of the said architecture parameter that is associated with the majority of the attribute descriptors included in the cluster. To predict the value of p_i for a new benchmark, B^l, we calculate the distance of its attribute vector to the centroid of all clusters, $\delta_i^l = [\delta_1^l, \dots \delta_q^l]$, and denote the closest cluster and the value associated with it by $\widehat{\delta_j^l}$ and $\widehat{v}_j$, respectively. To avoid the local minimum, we repeat this process for 10,000 times and assign the $\widehat{v}_j$ value adhered by the majority of the repetitions to p_i of the new benchmark. We use Matlab's *k-means* function from statistic toolbox in our model. A sample code implemented in Matlab is presented in Appendix A.4.

7.1.5 Classification

As an alternative method to clustering we use the classifier reported in [35]. The classifier is very similar to the regression NN ensemble presented in Chapter 4, Section 4.1.4. Both models use the same structure and heuristics but they use different error metrics. For example, as we discussed on page 61, the sensitivity heuristic exposes the set of trained neural networks obtained earlier to the perturbed exemplars. In the regression model, it discards the ones that their output largely affected by small input perturbations. In the classifier model, it discards the NNs that the predicted class changes for small input perturbations. More details can be found in [35].

In this particular problem, we develop a multi-classification model as to each code-attribute vector is assigned a configuration comprising several architectural parameters. Therefore, we develop 7 models, one for each architectural parameters.

7.2 Methodology

In this section we present more details on the different steps the proposed model takes to predict the optimum configuration for each GPGPU application.

As we discussed in the beginning of this chapter, the model trains on the attributes and optimal configurations of the benchmarks in the training set and the trained model is used to determine the optimal configuration of an arbitrary benchmark based on its attributes. The attributes of each benchmark are obtained using the SASSI instrumentation tool from NVIDIA [84]. We run the instrumented benchmarks on NVIDIA's Quadro K420 GPU. It is worth mentioning that the architecture of the GPU that runs the instrumented benchmark is irrelevant as we only use attributes of the benchmarks that are specific to the kernel code itself not the architecture of the GPU. The optimum configurations are obtained using the Pareto Front algorithm as discussed in Chapter 6. Each configuration is described by a vector of seven architectural parameters, i.e. number of memory controller blocks (MCB), Register File (RF) size, Shared Memory (SM) size, constant (Cnst), data level one (DL1), data level 2 (DL2) and instruction cache (IL1) sizes.

As we mentioned earlier in this chapter, we divide the range of each architectural parameter into a number of sub-ranges. This coarse-quantizes the configurations space to compensate for the small number of available benchmarks used in this study (24). Thus, each configuration parameter is characterized as "large", "medium" and "small" if the model uses three sub-ranges, or "large" and "small" if it uses two sub-ranges. The descriptors of the obtained Pareto-Optimal configurations of the said benchmark are then mapped onto the above mentioned coarse quantized space.

The sub-ranges are defined using the minimum and maximum values of the parameters over all the benchmarks. For example, if for benchmark A, the number of MCBs varies from 1 to 3, and for benchmark B it varies from 4 to 8, the overall range is from 1 to 8.

We represent each GPU configuration with a Super-Configuration which is formed by replacing the architectural parameters by the sub-ranges they belong to. Therefore, each Super-Configuration represents a number of GPU configurations. This will reduce the size of the configuration space the model needs to explore at the final step by decreasing the granularity of the architectural parameters.

We develop a model using the benchmarks in the training set and use the model to predict the performance-optimum configuration for an unknown benchmark based on the similarity of its code attributes to those which are used in the training process.

As mentioned in Section 7.1.2, we had a small set of 24 benchmarks (listed in Table 7.2) to evaluate our approach. We used leave-one-out cross validation where we trained the model on 23 of the 24 benchmarks and reported the model's performance

on the 24th benchmarks. We have used either K-Means clustering (C.F. Section 7.1.4) or classification (C.F. Section 7.1.5) to develop our models.

For each benchmark, we used a vector of 17 attributes obtained through profiling the benchmark as discussed in Section 7.1.3 and associated the Pareto Front super-Configuration as presented above and detailed in Section 7.2.1.

7.2.1 Obtaining the Optimum Configurations from Predicted Super-Configuration

The resulting configuration classes (Super-Configuration) represent a number of actual configurations, as shown in Figure 7.3 on page 95. For example, in our case, there are 7 configuration descriptors. If each configuration range includes 2 elements, then the maximum number of configurations represented by the chosen Super-Configuration is $2^7 = 128$. Table 7.2 shows the number of configurations in each predicted Super-Configuration for the chosen benchmark with 2 and 3 configuration ranges. On average, the size of the super-Configurations are 11% and 8% of the design space size for models with 2 and 3 configuration ranges. As we mentioned earlier in this section, because we select the ranges of the parameters over all the benchmarks, different benchmarks will have different number of configuration classes as well as components in each configuration class, based on their design space characteristics. In Table 7.2 we also report the number of configurations for the model with IL1 cache size removed from the configuration vectors. The reason is that through our experiments we realized the results of the model is sensitive to this parameter and we were interested to see how excluding the IL1 from configuration vectors would affect the performance of the model. We will discuss this in more detail in Section 7.3.3.

The next step in our process is to determine the Pareto-Optimal configuration among the configurations represented by the chosen Super-Configuration (see Figure 7.3). We determine this by either simulating the test code in all possible configurations in the selected Super-Configuration or by using our power-performance models [45].

The values reported in Table 7.1 are the attributes of the benchmarks and the values reported in Table 7.3 are the performance-optimum configuration for each benchmark along with the power and performance values associated to each configuration. Next to each parameter we report the the sub-ranges associated to each parameter in parentheses for the model with 3 sub-ranges. We normalize the attributes and scale

Table 7.2: The number of configurations in each predicted Super-Configuration for different number of sub-ranges.

benchmark	2 sub-ranges	3 sub-ranges	2 sub-ranges excluding IL1	total configs
NN	16	12	22	641
LPS	192	144	192	1280
STO	24	12	36	432
CP	48	18	48	3432
AES	228	96	228	23040
RAY	48	48	48	720
BFS	24	6	24	576
NQU	2	2	2	8
Needle	235	135	235	760
HotSpot	336	336	336	2012
BackProp	504	324	504	6897
SRAD	504	558	504	11541
VecAdd	120	90	120	720
Gaussian	20	20	20	120
DWT	210	70	210	1470
Transpose	64	42	64	240
EigenValues	18	18	18	324
ConvolutionTexture	64	42	64	768
MergeSort	388	196	383	4222
ConvolutionSeparable	3648	1008	3648	11842
LIB	36	72	36	540
MUM	36	294	36	3360
ScalarProd	288	294	288	7547
Histogram	388	196	388	4193

them to numbers between -1 and +1 before applying clustering or classification.

Table 7.3: Performance-optimum configurations and the corresponding sub-range numbers.

benchmark	number of MCBs	RF size	SM size	Cnst size	DL1 size	DL2 size	IL1 size	Prf (IPC)	Power (Watt)
NN [6]	7(3)	8KB(1)	8KB(1)	8KB(3)	16KB(1)	32KB(1)	2KB(1)	34	67
LPS [28]	5(2)	32KB(3)	8KB(1)	8KB(3)	8KB(1)	8KB(1)	2KB(1)	600	83
STO [1]	5(2)	32KB(3)	64KB(3)	8KB(3)	8KB(1)	4KB(1)	32KB(3)	238	81
CP [86]	6(3)	8KB(1)	6KB(1)	4KB(2)	16KB(1)	1KB(1)	4KB(1)	451	77
AES [58]	6(3)	32KB(3)	64KB(3)	4KB(2)	4KB(1)	3KB(1)	6KB(1)	655	95
RAY [60]	5(2)	32KB(3)	8KB(1)	8KB(3)	32KB(1)	8KB(1)	8KB(1)	484	133
BFS [8]	8(3)	32KB(3)	8KB(1)	8KB(3)	256KB(3)	256KB(3)	2KB(1)	136	63
NQU [75]	2(1)	32KB(3)	8KB(1)	8KB(3)	8KB(1)	8KB(1)	1KB(1)	34	22
Needle [8]	7(3)	2KB(1)	4KB(1)	8KB(3)	8KB(1)	256KB(3)	2KB(1)	11	16
HotSpot [8]	3(1)	32KB(3)	16KB(1)	8KB(3)	32KB(1)	16KB(1)	2KB(1)	506	114
BackProp [8]	7(3)	32KB(3)	8KB(1)	8KB(3)	64KB(2)	32KB(1)	1KB(1)	404	111
SRAD	7(3)	32(3)	32(2)	8(3)	32(1)	128(2)	4(1)	362	91
VecAdd	8(3)	16(2)	8(1)	8(3)	8(1)	16(1)	2(1)	178	124
Gaussian	3(1)	4(1)	8(1)	8(3)	8(1)	4(1)	1(1)	115	34
DWT	7(3)	32(3)	8(1)	8(3)	8(1)	32(1)	1(1)	531	138
Transpose	3(1)	8(1)	8(1)	8(3)	8(1)	0.5(1)	0.5(1)	3	20
EigenValues	8(3)	16(2)	64(3)	8(3)	8(1)	8(1)	0.5(1)	6	25
ConvolutionTexture	8(3)	16(2)	8(1)	8(3)	8(1)	64(2)	2(1)	648	114
MergeSort	5(2)	32(3)	32(2)	8(3)	64(2)	64(1)	2(1)	222	52
ConvolutionSeparable	7(3)	32(3)	32(2)	8(3)	16(1)	32(1)	8(1)	307	82
LIB	4(2)	16(2)	8(1)	8(3)	16(1)	16(1)	2(1)	11	63
MUM	7(3)	16(2)	8(1)	8(3)	32(1)	256(3)	4(1)	67	113
ScalarProd	7(3)	16(2)	16(1)	8(3)	128(3)	256(3)	1(1)	378	179
Histogram	3(1)	32(3)	32(2)	8(3)	64(2)	32(1)	2(1)	156	50

7.2.2 Evaluation

We use three methods to evaluate the accuracy of the model.

Distance

We calculate the distance of a configuration (A) to the actual optimum configuration (Act), and normalize it with respect to the actual configuration in the performance-power space as follow:

$$Normalized_Distance = \sqrt{\frac{(IPC_A - IPC_{Act})^2 + (Power_A - Power_{Act})^2}{(IPC_{Act})^2 + (Power_{Act})^2}} \quad (7.1)$$

Ranking

We rank the obtained configuration compared to all possible configurations in the design space. To this effect, we obtained the distance of each configuration in the design space from the performance-optimum one and then ranked the model-obtained configuration. We report the *ranking score* of the obtained configuration as the percentage of configurations that have distances smaller than the one of the obtained configuration.

Random

We also compare the results produced by the model to randomly selected configurations. To select the random configurations, we randomly select the same number of original configurations that the Super-Configuration represents. Then we apply the Pareto Optimal optimization and choose the one with the highest performance and compare it to the actual optimum configuration. We repeat the random experiment a number of times to satisfy 0.01% confidence interval and 95% confidence level. We report the average of the relative errors obtained as well as the percentage of cases that the model yields better results than the random experiment.

7.3 Experimental Setup and Results

In this section we present the results of the model for both K-Means and Classification techniques.

7.3.1 K-Means Clustering

Table 7.4 reports the ranking score of the predicted optimal configurations.

As the table shows, The model could predict the optimum configurations for 10 and 6 benchmarks for models with 2 and 3 clusters, respectively. For most of the others, the predicted configuration has power and performance close to the optimum configuration. On average, around 9% and 14% of the total configurations have better power-performance score than the predictions made by the models with 2 and 3 clusters, respectively. Part of the error of the model is due to some of the architectural parameters of some of the benchmarks that have unique values. For example, STO is the only benchmark with 32-KB IL1 cache size and its performance

Table 7.4: Ranking score of the selected optimal configurations using K-means clustering method.

benchmark	2 sub-ranges 2 clusters	3 sub-ranges 3 clusters	2 sub-ranges 2 clusters excluding IL1
NN	1.72%	1.72%	1.72%
LPS	0.00%	0.00%	0.00%
STO	36.57%	61.81%	3.24%
CP	22.93%	53.32%	22.93%
AES	0.04%	0.41%	0.04%
RAY	0.00%	0.00%	0.00%
BFS	17.97%	42.19%	17.97%
NQU	0.00%	0.00%	0.00%
Needle	3.42%	2.05%	3.42%
HotSpot	0.00%	0.00%	0.00%
BackProp	0.00%	0.04%	0.00%
SRAD	0.03%	0.90%	0.03%
VecAdd	0.14%	0.14%	0.14%
Gaussian	0.00%	0.00%	0.00%
DWT	0.00%	0.00%	0.00%
Transpose	59.58%	90.00%	59.58%
EigenValues	62.96%	62.96%	62.96%
ConvolutionTexture	2.08%	18.10%	2.08%
MergeSort	0.00%	1.40%	0.00%
ConvolutionSeparable	0.00%	0.41%	0.00%
LIB	0.19%	0.19%	0.19%
MUM	14.88%	13.04%	14.88%
ScalarProd	0.04%	0.52%	0.04%
Histogram	0.00%	0.19%	0.00%
Average	9.27%	14.56%	7.88%

is sensitive to this parameter. Therefore, we report in Table 7.4 the ranking scores of the model without IL1 cache parameter to see how making the parameter ranges more balanced affects the performance of the model. In Section 7.3.3 we will analyze these results in more details.

We have also compared the optimum configurations predicted by the model to to the randomly selected configurations using the method discussed in Section 7.2.2. The models with 2 and 3 sub-ranges, on average, produce predictions better that the random method in 68% and 52% of the predictions. We believe these results will be improved if we use larger set of benchmarks to build the model with.

7.3.2 Classification

Table 7.5 shows the ranking scores for the multi-classification models with different number of classes. The model could predict the optimum configuration for 9 and 7 benchmarks for the models that use 2 and 3 classes, respectively. Comparing two models, on average, the multi-classifier model makes more accurate results, specially when the number of sub-ranges are larger and the space is sparser. On average, the multi-classifier models with 2 and 3 sub-ranges produce better results than the random method in 70% and 60% of predictions, respectively. For a detailed analysis of the results please refer to Section 7.3.3.

7.3.3 Analysis

Although both models produce acceptable predictions for most of the benchmarks, for some, e.g. STO, CP, BFS, Transpose and EigenValues, the ranking scores are high. The main reason for the high scores is the small data set used to develop the model. The other reason is that for some of the parameters the number of benchmarks in each sub-range is imbalanced. For example, as reported in Table 7.3, for IL1 cache, only the STO's IL1 size is in sub-range 3 and all other benchmarks are categorized in sub-range 1. This is true for DL1 and constant cache as well. This has a significant impact on the accuracy of the model as there will be not enough exemplars in some of the sub-ranges. For example, the model miss-predicts the shared memory and instruction cache size of the STO benchmark because STO's instruction cache sub-range is unique and no other benchmark share that sub-range with STO. Also, only two other benchmarks have the same shared memory size as STO has. As the Power and performance of STO is sensitive to the instruction cache size, the prediction

Table 7.5: Ranking score of the selected optimal configurations using classification method.

benchmark	2 sub-ranges 2 classes	3 sub-ranges 3 classes	2 sub-ranges 2 classes excluding IL1
NN	2.65%	6.40%	0.78%
LPS	0.00%	0.31%	0.00%
STO	28.94%	62.96%	2.78%
CP	0.12%	25.87%	0.12%
AES	0.04%	0.43%	0.04%
RAY	0.00%	0.00%	0.00%
BFS	17.97%	42.97%	17.97%
NQU	0.00%	0.00%	0.00%
Needle	8.34%	27.22%	8.34%
HotSpot	0.00%	0.00%	0.00%
BackProp	0.00%	0.05%	0.00%
SRAD	0.02%	0.66%	0.02%
VecAdd	0.00%	0.00%	0.00%
Gaussian	0.00%	0.00%	0.00%
DWT	1.22%	1.22%	1.22%
Transpose	60.00%	92.50%	60.00%
EigenValues	95.37%	57.41%	95.37%
ConvolutionTexture	10.81%	2.47%	10.81%
MergeSort	0.00%	1.35%	0.00%
ConvolutionSeparable	0.06%	0.41%	0.06%
LIB	0.37%	0.00%	0.37%
MUM	0.09%	0.00%	0.09%
ScalarProd	0.03%	0.48%	0.03%
Histogram	0.00%	0.19%	0.00%
Average	9.42%	13.45%	8.25%

error is high. For BFS, the model miss-predicts the size of DL1 and DL2 caches. The optimum configuration for BFS has the largest DL1 and DL2 caches among all other benchmarks. Especially, its 256KB DL1 cache size is unique and the power and performance of the benchmark is sensitive to its size. For EigenValues, the model miss predicts the register file size and the shared memory size. This benchmark is one of three benchmarks with Shared memory size of sub-range 2. For Transpose, the model miss predicts the number of memory controller blocks and the register file size. We are studying the possible reasons that may cause the miss predictions of these parameters.

To further study the impact of the parameters with imbalanced sub-ranges on the prediction results, we have excluded the instruction cache from the models and compared the results with the original models. Excluding the instruction cache from the models with 2 sub-ranges and predicting the remaining 6 parameters reduced the ranking score of the STO for the clustering model from 36.57% to 3.24%. For the classifier model it reduced the ranking score of the STO and NN benchmarks from 2.65% and 28.94% to 0.78% and 2.78%, respectively. The results are reported in Table 7.4 and Table 7.5. Excluding the IL1 from the models, increases the Super-Configuration size of the STO and NN slightly (see Table 7.2).

The other observation from Table 7.4 and 7.5 is that the error of the model is proportional to the number of sub-ranges. This was expected as the number of sub-ranges affects the search space of the problem. With larger number of sub-ranges, each benchmark will have more super-Configuration in their design space and each super-Configuration will include less number of actual configurations. The downside of having few sub-ranges is that the predicted super-Configuration will contain more actual configurations which the user of the model needs to simulate to find the best power-performance configuration. Therefore, the number of sub-ranges is a trade-off between the prediction accuracy and number of simulations required to be done by the user of the model.

Furthermore, we examined the architectural parameters and observed that there are correlations between some of the parameters. For example, the number of memory controller blocks and DL2 cache size, DL1 and DL2 cache sizes, IL1 cache size and shared memory size are correlated. We are working on a new multi-classifier model that takes into account the correlation between architectural parameters. We believe this could improve the accuracy of the model.

7.4 Conclusion and Discussion

In this chapter we proposed a method to predict the power-performance optimum configuration of a GPU based on code-specific attributes. We have used models based on K-Means and non-linear classification [35]. Our results have established that code specific attributes can be used to obtain power and performance and in fact to obtain power-performance optimum configurations for a GPU.

Our method was accurate for most of the benchmarks we tested (18/24). However, our method was less accurate in 6 of the cases mainly because of the sparsity of the training set. Specifically the mispredicted cases were outliers having configurations that were unique and not represented by the remaining benchmarks.

The model presented in this chapter, relies on the NN-based predictor discussed in Chapter 4 to obtain the application specific optimum GPU configuration. The training time of the model depends on the size and complexity of the problem. In the next chapter we will explore techniques to accelerate the run time of the NN-based predictor presented in Chapter 4.

Chapter 8

Conclusion and Future Work

GPGPU applications usually have large amount of parallelism and different applications have different memory and computational demands. To run an application efficiently on a GPU, one needs to explore the hardware design space of GPU based on memory and computational requirements of the application. We proposed a number of different models for efficient resource allocation and design space exploration for GPGPU applications.

The first model, utilizes the Plackett-Burman methodology and formulates a constraint optimization problem to find the optimal GPU design for different available chip real-estate budgets without resorting to exhaustive design space exploration.

The GPGPU design space explorer, uses NN-based power and performance predictor, a filtering mechanism and Pareto Optimal optimization technique to predict application-specific power-performance optimum configurations. We showed that the optimum configurations produced by the model is very close to ones produced using simulation results. We proposed a method to obtain the configuration that performs well for any set of applications. The execution time of the proposed model was significantly reduced by formulating the time consuming part of its training phase in form of a quadratic optimization problem and solving it using quantum annealing techniques.

We also studied the influence of kernel attributes on the decisions made at architecture level and showed that the optimum configuration is influenced by kernel attributes. We developed a model to relate the power-performance optimum configuration of GPGPU benchmarks to the static kernel attributes.

The models presented in this work can be used by a) users of GPGPU applications to determine the power-performance optimum GPU configurations, b) GPU manu-

facturers to decide on the configuration of the next generation of GPUs, software developers to obtain insight on optimization opportunities to improve the efficiency of running applications on GPUs.

8.1 Future Direction

Here we present a few ideas and suggestions for future work.

1. Running a benchmark on the configurations predicted by the proposed models will result in optimum power-performance for the overall execution of the benchmark. Some benchmarks are composed of multiple kernels with different memory and computational requirements. The accuracy of the model can be improved by studying each kernel and determine the ones that the power and performance are most sensitive to.

   The execution of a kernel can be composed of different phases each of which with its own unique memory and computational demands. Predicting the optimum configurations based on the requirements of the major phases of the kernel can also improve the performance of the proposed models.

2. The model that could predict the optimum configuration at kernel or phase granularity (instead of application granularity), can be used along side power gating technique to adjust the GPU configuration at run time based on the demands of the major phases of the kernel.

3. The methods presented in this work were evaluated on a Fermi-like GPU architecture. We used the Fermi architecture since the tools available for it were reliable and mature. We believe that similar results could be held should the experimentation were to target one of the newer architectures. NVIDIA's newer architecture can run multiple kernels on GPU simultaneously. It is interesting to see how the proposed models can be applied to such environment.

4. In the model we proposed to predict the optimum power-performance configuration for a GPGPU application, we assumed the software parameters of the application are set and fixed. A model, based on the same methodology, can be developed to explore the software design space of an application to predict the best software configurations for a certain GPU that result in optimum power and performance.

5. The NN-based model is a feed-forward neural network. It is interesting to see how other neural network architectures such as Radial Basis Function (RBF) networks perform when they are used to build the model.

6. The optimum configurations predicted by our models can be translated into software optimization hints for the programmer of the system. For example, if the optimum configurations for a given benchmark have high number of memory controller blocks, it means the application accesses the off-chip DRAM frequently and therefore, using software managed cache to reduce the off-chip traffic can be beneficial to improve the performance.

Appendix A

Additional Information

A.1 Integer Linear Programming

In Chapter 3 we proposed to formulate GPGPU resource allocation as Knapsack optimization problem and solved it using Integer Linear Programming (ILP) technique. In this section we present the Matlab source code used in the model. The code below shows the source code for BFS benchmark with three architectural parameters.

```
% The value of each parameter
p=-[216145 6535 28065];
% The cost of each parameter
w=[0.3 2.71875  21.25];
% Initial number of units for each parameter
x0=[1 1 1];

f=[p(1) p(2) p(3)];
A=[w(1) w(2) w(3)];


lb=[1 1 1];
ub=[4 2 2];

Final_Bound_For_C=50;
index=1;

% main loop
for b=sum(w):0.3:Final_Bound_For_C
  [x,fval,exitflag,output,lambda] = linprog(f,A,b,[],[],lb,ub,x0);
  x_continous(:,index)=x;
```

```
    x_final(:,index)=round(x);
    index=index+1;
end

cost=x_final'*w';


% Plot the results
hold on
plot(sum(w):0.3:Final_Bound_For_C,x_final(1,:),'linewidth',2)
plot(sum(w):0.3:Final_Bound_For_C,x_final(2,:),'r','linewidth',2)
plot(sum(w):0.3:Final_Bound_For_C,x_final(3,:),'k','linewidth',2)

axis([24 Final_Bound_For_C 0 5])
legend('MCB','DL1.Cache','RF')

xlabel('Cost (transistor budget)','fontsize',14)
ylabel('Configurations','fontsize',14)
grid on
```

Listing A.1: Solving Knapsack optimization problem using ILP.

Figure A.1 shows the best configuration per cost for BFS benchmark generated by running the code above under the Matlab.

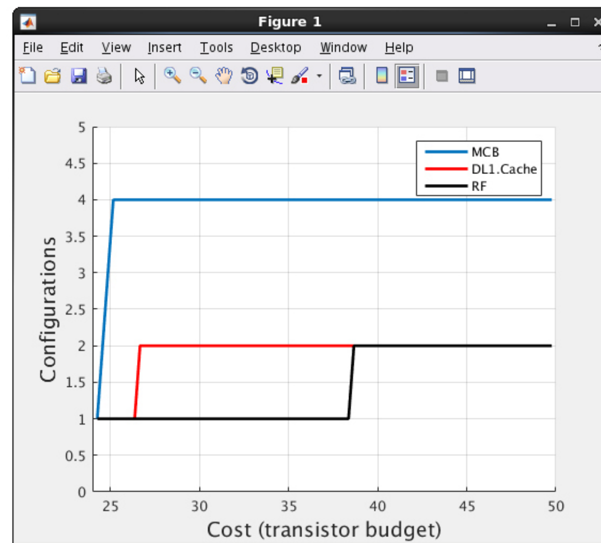


Figure A.1: The optimum resource allocation for BFS benchmark.

A.2 Outlier Detection Heuristic

In Chapter 4 we proposed a heuristic method to filyet extreme outliers. The code below is the source code of the filter implemented in Matlab.

```
clear all;
clc;

% read inputs
All_Data;
Blind_res;
BlindData;
data_All_file;
load('Set.mat');
load('Seq_Indexes');

% Initial parameters -- application specific
Neighbours_T=4;
data_blind=data_all_2;
TrainSet_Complete;
Y=data_blind(:,1:8); % Normalized Test Set.
IPC_Error_Col=18; % for performance
[tp, num_of_Dimensions] = size(Y);

num=[1:tp]';
AllData=[num, AllData];
[ADa, ADb]=size(AllData);
Pre_TestActiv=AllData(:,IPC_Error_Col-1)./10; % Predicted activities of
    the test set.
IPC_Error_Data=AllData(:,IPC_Error_Col);
Max_IPC_Error_Outlier=mean(IPC_Error_Data)+2*sqrt(var(IPC_Error_Data));
%Max_IPC_Error_Outlier=15;
Num_of_Outlier_IPC = sum(AllData(:,IPC_Error_Col)>Max_IPC_Error_Outlier)
    ;
Num_IPC_OutL_L60 = sum(AllData(:,IPC_Error_Col)>60);
Num_IPC_OutL_L50 = sum(AllData(:,IPC_Error_Col)>50) - Num_IPC_OutL_L60;
Num_IPC_OutL_L40 = sum(AllData(:,IPC_Error_Col)>40) - Num_IPC_OutL_L50 -
     Num_IPC_OutL_L60;
Num_IPC_OutL_L30 = sum(AllData(:,IPC_Error_Col)>30) - Num_IPC_OutL_L40 -
     Num_IPC_OutL_L50 - Num_IPC_OutL_L60;
[tp, num_of_Dimensions] = size(Y);
Data_Blind_Size = tp;
```

```
Filter_Book=[0.5 1 2 3 4 5 10 15 20 30 40 50];
All_Act_T_Mean=[];

% Main loop that checks all the sequences.
for k=1:size(Set,1);

    X_T=cell2mat(Set(k,4)); % Test set for SeqX
```

Listing A.2: Outlier filter.

A.3 Pareto Front Optimization

In this section we present the Matlab code to obtain the Pareto Front configurations of an application.

```
clear all
clc
% Reading the inputs
Control=0;
Design_Space;
DS_IPC_Power=DS;
Actual=DS_IPC_Power(:,[8,9]) ;

% Preparing data for PF algorithm
Sim=[Actual(:,1)*-1 Actual(:,2)];

% Calling the function that calculates PF
[SimPar SimInd]=prtp(Sim);

Actual_Pareto= sortrows([SimPar(:,1)*-1 SimPar(:,2) SimInd'],-1);
Actual_Index=SimInd;

% Preparing data to plot
k=1;
j=1;
for i=1:size(DS_IPC_Power,1)
    if(~sum(ismember(Actual_Index,i)))
        Actual_Remaining_Index(j,:)=i;
        j=j+1;
    end
```

```
end

Actual_Remaining_Pareto=Sim(Actual_Remaining_Index,:);


if (Control==0)
     plot(Actual_Pareto(:,1),Actual_Pareto(:,2),'r*')
     hold on
     plot(Actual_Remaining_Pareto(:,1)*-1,Actual_Remaining_Pareto(:,2),'b
    +')
end


Complete_Actual_Pareto=DS_IPC_Power(SimInd,:);
Complete_Non_Actual_Pareto=DS_IPC_Power(Actual_Remaining_Index,:);

% Plotting the data
```

Listing A.3: Pareto Front algorithm.

A.4 Kmeans Clustering

In this section we present a Matlab sample code that clusters the input application code attribute vectors, assigns a IL1 cache size to each cluster based on majority vote system and based on the distance to the centroids of the clusters, predicts the IL1 cache size of a new application.

```
clear all
clc

% Reading input data
GPGPU_Dataset;

% Initialization
MCB = 18 ;
Reg = 19 ;
Shard = 20 ;
Constant = 21 ;
DL1 = 22 ;
DL2 = 23 ;
IL1 = 24 ;
```

```
CTA = 25 ;
KM_Rep=5;
Num_Descriptors = 17;
Config = IL1;
Repet=10000;
Num_Clusters=3;
ii=1;

% main loop
for rep = 1: Repet
    % rep for each application
    for kk=1:size(GPGPUDataSet,1)

        Blind_Index = kk;

        GPGPU_Dataset;

        GPUDataTargets = GPGPUDataSet(:,Config);
        % Strip the dataset out of labels and the indices
        GPUDataDescriptors = GPGPUDataSet(:,1:Num_Descriptors);

        [a,b]=size(GPUDataDescriptors);
    % normalize data
        GPUDataDescriptorsNormalized=2*(GPUDataDescriptors- ones(a,1)*
    min(GPUDataDescriptors))./(ones(a,1)*(max(GPUDataDescriptors)-min(
    GPUDataDescriptors)))-1;
        GPUDatasetNormalized = [ GPUDataDescriptorsNormalized ,
    GPUDataTargets ];

        Train_Index=setdiff(1:size(GPGPUDataSet,1),Blind_Index);
```

Listing A.4: Kmeans clustering.

Bibliography

[1] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM.

[2] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 725–737, New York, NY, USA, 2015. ACM.

[3] K. Audhkhasi, O. Osoba, and B. Kosko. Noise benefits in backpropagation and deep bidirectional pre-training. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Aug 2013.

[4] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. *SIGPLAN Not.*, 45(5):105–114, January 2010.

[5] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163 –174, april 2009.

[6] Billconan and Kavinguy. A neural network on gpu. http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU. [Online; accessed 19-April-2017].

[7] M. Blott. Reconfigurable future for HPC. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 130–131, July 2016.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[9] L. Chua. Memristor-The missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, September 1971.

[10] E. Cohen and B. Tamir. Quantum annealing - foundations and frontiers. *The European Physical Journal Special Topics*, 224(1):89–110, Feb 2015.

[11] AMD Corporation. ATI Radeon HD 5000 series: an inside view. 2010.

[12] Nvidia Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2009.

[13] NVIDIA Corporation. NVIDIA's Next Generation CUDATM Compute Architecture: Kepler GK110 whitepaper. 2012.

[14] NVIDIA Corporation. Whitepaper: NVIDIA Tesla P100. 2016.

[15] NVIDIA Corporation. Whitepaper: NVIDIA TESLA V100 GPU ARCHITECTURE. 2017.

[16] Daniel Crawford, Anna Levit, Navid Ghadermarzy, Jaspreet S. Oberoi, and Pooya Ronagh. Reinforcement Learning Using Quantum Boltzmann Machines. *1QBit publications*, page 17, 2016.

[17] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[18] S. Dublish, V. Nagarajan, and N. Topham. Characterizing memory bottlenecks in GPGPU workloads. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–2, Sept 2016.

[19] Krzysztof Dudziński and Stanisław Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28(1):3 – 21, 1987.

[20] A. Duran and M. Klemm. The Intel Many Integrated Core Architecture. In *2012 International Conference on High Performance Computing Simulation (HPCS)*, pages 365–366, July 2012.

[21] Bishwajit Dutta, Vignesh Adhinarayanan, and Wu-chun Feng. GPU Power Prediction via Ensemble Machine Learning for DVFS Space Exploration. *Computer Science Technical Report, Department of Computer Science, Virginia Polytechnic Institute and State University*, 2018.

[22] the quantum computing company DWave. The D-Wave 2X Quantum Computer, Technology Overview. https://www.dwavesys.com/sites/default/files/D-Wave2XTechCollateral0915F.pdf, 2015. [Online; accessed 19-July-2018].

[23] S. W. Ellacott. *An Analysis of the Delta Rule*, pages 956–959. Springer Netherlands, Dordrecht, 1990.

[24] Marshall L Fisher. Worst-case analysis of heuristic algorithms. *Management Science*, 26(1):1–17, 1980.

[25] GLOVER FRED. Tabu Search - Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[26] GLOVER FRED. Tabu Search - Part II. *ORSA Journal on Computing*, 2(1):4–33, 1990.

[27] W.W.L. Fung, I. Sham, G. Yuan, and T.M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 407 –420, dec. 2007.

[28] Mike Giles. Jacobi iteration for a Laplace discretisation on a 3D structured grid. https://people.maths.ox.ac.uk/gilesm/codes/laplace3d/laplace3d.pdf, 2008. [Online; accessed 17-Sep-2018].

[29] J. Gosling. *Introductory Statistics*. Quicksmart University Guides Series. Pascal Press, 1995.

[30] Khronos Group. Open Computing Language (OpenCL). https://www.khronos.org/opencl/. [Online; accessed 23-April-2018].

[31] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, Oct 1990.

[32] Mark Harris. Mapping Computational Concepts to GPUs. In series editor Matt Pharr, Randima Fernando, editor, *GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation*, pages 493–508. Addison Wesley, 2005.

[33] Mark Harris. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/, 2017. [Online; accessed 2-Jul-2018].

[34] B.K. Hedayati and et. al. An improved neural network ensemble model of Aldose Reductase inhibitory activity. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–7, 2012.

[35] Keshavarz Hedayati and Nikitas Dimopoulos. Sensitivity and Similarity Regularization in Dynamic Selection of Ensembles of Neural Networks. *Accepted: The International Joint Conference on Neural Networks.*, 2017.

[36] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[37] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[38] Jen-Hsun Huang. The opening keynote speech at the GPU Technology Conference. https://www.theregister.co.uk/2013/03/19/nvidia_gpu_roadmap_computing_update, 2013. [Online; accessed 12-July-2018].

[39] Christopher J. Hughes. *Single-Instruction Multiple-Data Execution*. Morgan and Claypool, 2015.

[40] Intel. The Compute Architecture of Intel Processor Graphics Gen 9. https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-

Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf, 2015. [Online; accessed 29-Jun-2018].

[41] Wenhao Jia, Elba Garza, Kelly A. Shaw, and Margaret Martonosi. GPU Performance and Power Tuning Using Regression Trees. *ACM Trans. Archit. Code Optim.*, 12(2):13:1–13:26, May 2015.

[42] Wenhao Jia, K.A. Shaw, and M. Martonosi. Stargazer: Automated regression-based GPU design space exploration. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 2 –13, april 2012.

[43] Mark W Johnson, Mohammad HS Amin, Suzanne Gildert, Trevor Lanting, Firas Hamze, Neil Dickson, R Harris, Andrew J Berkley, Jan Johansson, Paul Bunyk, et al. Quantum annealing with manufactured spins. *Nature*, 473(7346):194–198, 2011.

[44] A. Jooya, N. Dimopoulos, and A. Baniasadi. GPU design space exploration: NN-based models. In *Communications, Computers and Signal Processing (PACRIM), 2015 IEEE Pacific Rim Conference on*, pages 159–162, Aug 2015.

[45] A. Jooya, N. Dimopoulos, and A. Baniasadi. MultiObjective GPU design space exploration optimization. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 659–666, July 2016.

[46] A. Jooya, N. Dimopoulos, and A. Baniasadi. Optimum Power-Performance GPU Configuration Prediction Based on Code Attributes. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 418–425, July 2017.

[47] Ali Jooya, Amirali Baniasadi, and Nikitas J. Dimopoulos. Efficient design space exploration of GPGPU architectures. In *Proceedings of the 18th international conference on Parallel processing workshops*, Euro-Par'12, pages 518–527, Berlin, Heidelberg, 2013. Springer-Verlag.

[48] Ali Jooya, Babak Keshavarz, Nikitas Dimopoulos, and Jaspreet S. Oberoi. Accelerating Neural Network Ensemble Learning Using Optimization and Quantum Annealing Techniques. In *Proceedings of the Second International Work-*

shop on Post Moores Era Supercomputing, PMES'17, pages 1–7, New York, NY, USA, 2017. ACM.

[49] B. Keshavarz-Hedayati, P. Guangyuan, A. Jooya, and N. J. Dimopoulos. In-Training and Post-Training Generalization Methods: the case of ppar$-\alpha$ and ppar$-\gamma$ agonists. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, July 2015.

[50] Ahmad Lashgar and Amirali Baniasadi. Performance in GPU Architectures: Potentials and Distances. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking, 2011. WDDD 2011. Held in conjunction with the 38th International Symposium on Computer Architecture (ISCA-38)*, June 2011.

[51] E.M. Laxdal, R. Parra-Hernandez, and N.J. Dimopoulos. Guided construction of training data set for neural networks. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 6, pages 5905–5910 vol.6, Oct 2004.

[52] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[53] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators. *SIGARCH Comput. Archit. News*, 39(3):129–140, June 2011.

[54] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39 –55, march-april 2008.

[55] A. Lopes, F. Pratas, L. Sousa, and A. Ilic. Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 259–268, April 2017.

[56] J. Macqueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[57] Souley Madougou, Ana Varbanescu, Cees de Laat, and Rob van Nieuwpoort. The Landscape of GPGPU Performance Modeling Tools. *Parallel Comput.*, 56(C):18–33, August 2016.

[58] S.A. Manavski. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68, Nov 2007.

[59] S Mandelbrojt and L Schwarts. Jacques Hadamard (1865–1963). *Kabul Amer. Math. Soc.*, 71:107–129, 1965.

[60] Maxime. Ray tracing. http://www.nvidia.com/cuda. [Online; accessed 19-April-2017].

[61] Sparsh Mittal and Jeffrey S. Vetter. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, August 2014.

[62] G. E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sept 2006.

[63] G. Narzisi. Multi-objective Optimization. http://cims.nyu.edu/ gn387/glp/lec1.pdf. [Online; accessed 17-Sep-2018].

[64] NVIDIA. CUDA Code Samples. https://developer.nvidia.com/cuda-code-samples. [Online; accessed 19-April-2017].

[65] NVIDIA. CUDA Zone. https://developer.nvidia.com/cuda-zone. [Online; accessed 23-April-2018].

[66] NVIDIA. NVIDIA Corporation. NVIDIA CUDA SDK code samples.

[67] NVIDIA. NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110, White Paper. 2012.

[68] NVIDIA. NVIDIA CUDA C Programming Guide, Version 9.2. NVIDIA, 2018.

[69] NVIDIA. Parallel Thread Execution ISA Version 6.2. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2018. [Online; accessed 6-June-2018].

[70] OpenACC.org. Open Accelerators (OpenACC). https://www.openacc.org/. [Online; accessed 23-April-2018].

[71] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[72] G. Palermo, C. Silvano, and V. Zaccaria. ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, Dec 2009.

[73] R. Parra-Hernandez and N. J. Dimopoulos. A New Heuristic for solving the Multichoice Multidimensional Knapsack Problem (MMKP). *IEEE Trans. On Systems Man and Cybernetics PartA*, Vol 35(5):708–717, Sep 2005.

[74] R. Parra-Hernandez, E.M. Laxdal, N.J. Dimopoulos, and P. Alexiou. A new neural network ensemble heuristic for a predictor of the aldose reductase inhibitory activity. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 838–843, Dec 2005.

[75] Pcchen. N-Queen solver for CUDA. https://devtalk.nvidia.com/default/topic/397284/n-queen-solver-for-cuda/. [Online; accessed 19-April-2017].

[76] R. Plackett and J. Burman. Design of Optimum Multifactorial Experiments. *Biometrika*, 33(4):305–325, 1944.

[77] Russell Reed and Robert J Marks. *Neural smithing: supervised learning in feedforward artificial neural networks.* Mit Press, 1999.

[78] Gili Rosenberg, Mohammad Vazifeh, Brad Woods, and Eldad Haber. Building an Iterative Heuristic Solver for a Quantum Annealer. *Comput. Optim. Appl.*, 65:845–869, 2016.

[79] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen mei W. Hwu. Program optimization

space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[80] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[81] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[82] M. Steinhaus, R. Kolla, J. Larriba, T. Ungerer, and M. Valero. Transistor count and chip-space estimation of simplescalar based microprocessor models. In *Workshop on Complexity-Effective Design*, 2001.

[83] Lorenzo Stella and Giuseppe E Santoro. Quantum annealing of an Ising spin-glass by Green's function Monte Carlo. *Physical Review E*, 75(3):036703, 2007.

[84] M. Stephenson, S. K. S. Hari, and et. al. Flexible software profiling of GPU architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 185–197, June 2015.

[85] Mark Stephenson. Sassi user guide. pages 1–15, 2015.

[86] The-Impact-Research-Group. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil/parboil.aspx. [Online; accessed 19-April-2017].

[87] Top 500 list. https://www.top500.org. [Online; accessed 16-July-2018].

[88] P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[89] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

[90] R. S. Williams. What's Next? [The end of Moore's law]. *Computing in Science Engineering*, 19(2):7–13, Mar 2017.

[91] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[92] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *Micro, IEEE*, 31(2):50 –59, march-april 2011.

[93] Xilinx. Automotive-grade XA Spartan-6 LXT FPGAs. https://www.xilinx.com/products/silicon-devices/fpga/xa-spartan-6/lxt.html. [Online; accessed 29-May-2018].

[94] Xilinx. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. [Online; accessed 4-July-2018].

[95] Xilinx. LogiCORE IP Multi-Port Memory Controller. MPMC (v6.04.a), July 6 2011.

[96] J. J. Yi, D. J. Lilja, and D. M. Hawkins. Improving computer architecture simulation methodology by adding statistical rigor. *IEEE Transactions on Computers*, 54(11):1360–1373, Nov 2005.

[97] C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. In *2006 International Symposium on System-on-Chip*, pages 1–4, Nov 2006.

[98] Z. Yu, J. Wang, L. Eeckhout, and C. Xu. QIG: Quantifying the Importance and Interaction of GPGPU Architecture Parameters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1211–1224, 2017.

[99] Arman Zaribafiyan, Dominic JJ Marchand, and Seyed Saeed Changiz Rezaei. Systematic and deterministic graph-minor embedding for cartesian products of graphs. *arXiv preprint arXiv:1602.04274*, 2016.

[100] Yao Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382 –393, feb. 2011.

[101] Ying Zhang, Lu Peng, Bin Li, Jih-Kwon Peir, and Jianmin Chen. Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205 –215, nov. 2011.

[102] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms—a comparative case study. In *International conference on parallel problem solving from nature*, pages 292–301. Springer, 1998.