DOI:10.15774/PPKE.ITK.2014.001

Efficient implementation of computationally intensive algorithms on parallel computing platforms

Csaba Nemes

A thesis submitted for the degree of
Doctor of Philosophy

Scientific adviser:
Zoltán Nagy, PhD

Supervisor:
Péter Szolgay, DSc

Faculty of Information Technology And Bionics
Péter Pázmány Catholic University

Budapest, 2014

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

Acknowledgements

First of all, I would like to thank my scientific advisor Zoltán Nagy and my consultant Prof. Péter Szolgay for guiding and supporting me over the years.

I am equally thankful to Örs Legeza and Gergely Barcza for teaching me quantum physics. Furthermore, I would like to give an extra special thank to Gergely Barcza for supporting me as a friend and as the godfather of my son.

I am grateful to Prof. Tamás Roska, Prof. Árpád Csurgay and Judit Nyékyné Gaizler, PhD for giving me encouragement and the opportunity to carry out my research at the university.

Conversations and lunches with Balázs Varga and Ádám Balogh are highly appreciated. Realistic thoughts from my ex-colleague and friend Tamás Fülöp always helped me to accurately determine my position even without a GPS. The cooperation of my closest colleagues András Kiss, Miklós Ruszinkó, Árpád Csík, László Füredi, Antal Hiba and Endre László is greatly acknowledged.

Most importantly, I am thankful to my family. I wish to thank my wife Fruzsi and son Zente for their loving care and tolerating my frequent absence. Finally, I am lucky to have my parents, grandparents, and one great grandparent sponsoring my never-ending studies.

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

Contents

1 Introduction **1**

2 Parallel architectures **5**

2.1 Field Programmable Gate Arrays 7

2.1.1 The general structure 8

2.1.2 The common peripherals 11

2.1.2.1 General purpose I/Os 11

2.1.2.2 Transceiver I/Os 12

2.1.3 Xilinx Virtex-6 SX475T FPGA 13

2.2 Graphical Processing Units 13

2.2.1 NVidia Kepler architecture 15

2.2.1.1 The general structure 15

2.2.1.2 CUDA programming 16

2.2.1.3 NVidia K20 18

3 Solving Partial Differential Equations on FPGA **21**

3.1 Computational Fluid Dynamics (CFD) 21

3.1.1 Euler equations 21

3.1.2 Finite volume method solution of Euler equations 22

3.1.2.1 Structured mesh 23

3.1.2.2 Unstructured mesh 25

3.2 Data structures and memory access patterns 27

3.3 Structure of the proposed processor 29

3.4 Outline of the multi-processor architecture 31

3.5 Analysis of the chosen design strategies 32

DOI:10.15774/PPKE.ITK.2014.001

4 Generating Arithmetic Units: Partitioning and Placement 35
4.1 Locally distributed control of arithmetic unit 35
4.1.1 The proposed control unit 35
4.1.2 Trade-off between speed and number of I/Os 39
4.2 Partitioning problem 40
4.2.1 Problem formulation 40
4.3 Partitioning algorithms used in circuit design 43
4.3.1 Move-based heuristics 44
4.3.1.1 The Kernighan-Lin algorithm 44
4.3.1.2 The Fiduccia-Mattheyses algorithm 45
4.3.2 Spectral partitioning 46
4.3.2.1 Spectral bipartitioning 47
4.3.2.2 Spectral partitioning with multiple eigenvectors . . 48
4.3.3 Simulated annealing 49
4.3.4 Software packages incorporating the multilevel paradigm . . . 50
4.3.4.1 Chaco 51
4.3.4.2 hMetis 52
4.4 Empirically validating the advantage of locally controlled arithmetic units 53
4.4.1 The proposed greedy algorithm 54
4.4.1.1 Preprocessing and layering 56
4.4.1.2 Swap-based horizontal placement 58
4.4.1.3 Greedy partitioning based on spatial information . . 59
4.4.2 The configuration of the hMetis program 59
4.4.3 Comparison and evaluation 61
4.5 Partitioning and placement together 65
4.5.1 Properties of a good partition 65
4.5.2 The proposed algorithm 67
4.5.2.1 Preprocessing and Layering 67
4.5.2.2 Floorplan with simulated annealing 67
4.5.2.3 New representation for graph partitioning 69
4.5.2.4 Partitioning 72
4.5.2.5 Outline of the full algorithm 74

DOI:10.15774/PPKE.ITK.2014.001

4.5.2.6 Comparison to the terminal propagation technique . 74
4.5.3 Framework 76
4.5.4 Results 76
4.6 Summary 77

5 Density Matrix Renormalization Group Algorithm 81
5.1 Previous implementations 81
5.2 Investigated models 82
5.2.1 Heisenberg model 83
5.2.2 Hubbard model 83
5.3 Symmetries to be exploited 85
5.4 Algorithm 86
5.4.1 *LR strategy* 89
5.4.2 *l*-1-1-*r strategy* 90
5.5 Parallelism and run-time analysis 90
5.6 Limits of the FPGA implementation 93

6 Hybrid GPU-CPU acceleration of the DMRG algorithm 95
6.1 Accelerating matrix-vector multiplications 95
6.1.1 Architectural motivations 97
6.1.2 gemv_trans() 98
6.1.3 gemv() 103
6.2 Accelerating projection operation 103
6.2.1 Architectural motivations 105
6.2.2 Scheduling strategies 106
6.3 Implementation results 110
6.4 Summary 114

7 Summary of new scientific results 117
7.1 New Scientific Results 117
7.2. Új tudomanyos eredmenyek (in Hungarian) 121
7.2. Application of the Results 125

References 134

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

Structure of the dissertation

Chapter 1 introduces the motivation behind the scientific work presented in the dissertation.

In *Chapter 2*, frequently used parallel architectures are reviewed with emphasis on the FPGA and GPU architectures, which were used during my research.

In *Chapter 3*, the numerical solution of partial differential equations on FPGAs is discussed including two complex test cases, the description of the proposed architecture, and the analysis of the chosen design strategies. The chapter contains scientific results which belong to the research group I worked in, and provides the background information to the high-level design methodology (presented in the next chapter) which is my scientific contribution.

In *Chapter 4*, a new high-level design methodology is presented to design high-performance, locally controlled arithmetic units for FPGA. The chapter contains the scientific work related to my first thesis group.

Chapter 5 provides the background information to the hybrid CPU-GPU acceleration of the density matrix renormalization group algorithm (presented in the next chapter). It contains the description of the algorithm and the investigated models, which are part of the scientific literature, the run-time analysis of my CPU-only implementation, and my estimation of the performance of a possible FPGA acceleration.

In *Chapter 6* the first hybrid CPU-GPU acceleration of the density matrix renormalization group algorithm is presented including a new scheduling for the matrix operations of the projection operation and a new CUDA kernel for the asymmetric transposed matrix-vector multiplication. The chapter contains the scientific work related to my second thesis group.

Finally, in *Chapter 7*, the new scientific results of the dissertation are summarized, and the possible application areas are enumerated.

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

List of Figures

2.1 Schematic view of a simplified FPGA 9
2.2 Schematic of a simplified logic cell 9
2.3 A schematic block diagram of the Kepler GK110 chip 17

3.1 A simulation of the airflow inside a scramjet engine 23
3.2 Interface with the normal vector and the cells required in the computation 26
3.3 An unstructured mesh and the corresponding descriptors 28
3.4 Block diagram of the proposed processor 30
3.5 Outline of the proposed architecture 32

4.1 Schematic of the control unit 37
4.2 A partitioned data-flow graph and the corresponding cluster adjacency graph 38
4.3 Operating frequency of the proposed control unit. 40
4.4 Placed instances of a locally controlled AU produced with a naive partitioning technique 55
4.5 A simple data-flow garph and its layered version 57
4.6 The structured CFD graph partitioned by the proposed algorithm . . . 60
4.7 Placed instances of the locally controlled AU produced with the proposed partitioning 62
4.8 A fragment of the first belt of Figure 4.9 is shown to demonstrate how inheritance works. 70
4.9 The partitioned data-flow graph of the unstructured CFD problem . . 74
4.10 Operating frequency and area requirements of the AU as the maximum number of I/O connections of a cluster is changing. 77

DOI:10.15774/PPKE.ITK.2014.001

5.1 Exploiting the projection symmetry in the Heisenberg model 87
5.2 Exploiting the conservation of particle number in the spin-$\frac{1}{2}$ Hubbard model 87
5.3 Heisenberg model: The run-time analysis of the algorithm 91
5.4 Hubbard model: The run-time analysis of the algorithm 91

6.1 The memory request of gemv_trans() in case of the Heisenberg model 96
6.2 GTX 570: Performance of the presented *gemv_trans()* kernel 98
6.3 K20, no shuffle operation in the kernel: Performance of the presented *gemv_trans()* kernel 100
6.4 K20, shuffle operation enabled: Performance of the presented *gemv_trans()* kernel 101
6.5 Similar to Figure 6.4 but for matrix height $5e5$. 102
6.6 Similar to Figures 6.4 and 6.5 but for matrix height $1e6$. 102
6.7 Performance of the gemv_normal() operation 103
6.8 GPU memory footprints of the two strategies are compared in case of the Heisenberg model 105
6.9 Interleaved operation records and the resulting parallel execution . . . 108
6.10 GTX 570, Heisenberg model: Performance of the two strategies is compared. 109
6.11 Similar to Figure 6.10 but on K20 architecture. 109
6.12 GTX 570, Heisenberg model: Performance results of the hybrid CPU-GPU acceleration of the projection operation. 111
6.13 Similar to Figure 6.12 but for the Hubbard model on GTX 570. 111
6.14 Similar to Figures 6.12 and 6.13 but for the Heisenberg model on K20. 112
6.15 Similar to Figures 6.12, 6.13 and 6.14 but for the Hubbard model on K20. 112
6.16 K20, Heisenberg model: Acceleration of different parts of the algorithm is compared for $m = 4096$. 114
6.17 K20, Hubbard model: Acceleration of different parts of the algorithm is compared for $m = 4096$. 115

DOI:10.15774/PPKE.ITK.2014.001

List of Tables

2.1 Virtex-6 FPGA Feature Summary 14
2.2 NVIDIA Tesla product line and the GTX 570 GPU 16

4.1 Implementation results of different partitioning strategies in case of the 32 bit structured CFD problem. 63
4.2 Comparing operating frequency of the 32 bit and the 64 bit AU in case of the structured CFD problem. 63
4.3 Partitioning and implementation results of the structured CFD graph. . 78
4.4 Partitioning and implementation results of the unstructured CFD graph. 79

5.1 Virtex-7 XC7VX1140T FPGA feature summary 93

6.1 Runtime of the accelerated matrix-vector operations of the Davidson algorithm 104
6.2 Total time of strategies is compared 110
6.3 Heisenberg model: final timings compared 113
6.4 Hubbard model: final timings compared 113
6.5 Model comparison in case of Xeon E5 + K20. 113

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

List of Abbreviations

ASIC Application-specific Integrated Circuit

AU Arithmetic Unit

BLAS Basic Linear Algebra Subprograms

BRAM Block RAM

CFD Computational Fluid Dynamics

CLB Configurable Logic Blocks

CMOS Complementary metal–oxide–semiconductor

CNN Cellular Neural/Nonlinear Network

CNN-UM Cellular Neural/Nonlinear Network - Universal Machine

CPU Central Processing Unit

CRS Compressed Row Storage

CU Control Unit

CuBLAS CUDA BLAS

CUDA Compute Unified Device Architecture

DDR3 Double Data Rate type three SDRAM

DDR4 Double Data Rate type four SDRAM

DMA Direct Memory Access

DOI:10.15774/PPKE.ITK.2014.001

DMRG Density Matrix Renormalization Group

DSP Digital Signal Processing

EC Edge Coarsening

FIFO First In, First Out

FIR Finite Impulse Response

FM Fiduccia-Mattheyses algorithm

FPGA Field Programmable Gate Array

FPU Floating-point Unit

FVM Finite Volume Method

GDDR5 Graphics Double Data Rate type five SDRAM

GPU Graphical Processing Unit

HC Hyperedge Coarsening

I/O Input/Output

IDE Integrated Development Environment

IP Intellectual Property

KL Kernighan-Lin algorithm

LUT Look-up Table

MAC Media Access Control

MACC Multiply Accumulate

MADD Multiply Add

MHC Modified Hyperedge Coarsening

MKL Math Kernel Library

DOI:10.15774/PPKE.ITK.2014.001

MPI Message Passing Interface

MUX Multiplexer

NIC Network Interface Controller

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PCI Peripheral Component Interconnect

PCIe PCI Express

PDE Partial Differential Equation

RAM Random-access Memory

RDMA Remote Direct Memory Access

RTL Register-transfer-level

SA Simulated Annealing

SATA Serial Advanced Technology Attachment

SDK Software Development Kit

SDRAM Synchronous Dynamic RAM

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SM Streaming Multiprocessor

SMX Kepler Streaming Multiprocessor

SoC System-on-Chip

SP2 Second-order Spectral Projection algorithm

SRAM Static RAM

DOI:10.15774/PPKE.ITK.2014.001

SRAM Static Random Access Memory

STL Standard Template Library

TPS Tensor Product States

UCF User Constraint File

VHDL VHSIC Hardware Description Language

DOI:10.15774/PPKE.ITK.2014.001

Chapter 1

Introduction

Computationally intensive simulations of physical phenomena are inevitable to solve engineering and scientific problems. Simulations are used to test product designs without fabrication or to predict properties of new physical or chemical systems. Computer engineering has long since been dealing with the acceleration of simulations to decrease the development time of new products or to improve the resulting quality by expanding the design space. Since clock frequency of processors reached the physical limits caused by power dissipation, processor designers are focusing on multi- and many-core architectures to keep up with the predictions of Moore's law. The goal of high-performance computing is to answer how to exploit the computing potential of these novel parallel architectures, such as GPU (*Graphical Processing Unit*) and FPGA (*Field Programmable Gate Array*), to solve computationally intensive problems.

During my research I investigated the acceleration of two specific problems with the following questions in mind: What is the best architecture for the given application? How can the implementation methodology be improved? What performance can be reached, and what are the implementation tradeoffs in terms of speed, power and area?

The first problem I investigated was the numerical solution of *partial differential equations* (PDEs) on FPGAs. Nagy et al. demonstrated that the FPGA implementation of the emulated digital CNN-UM (*Cellular Neural Network - Universal Machine*) can be generalized to efficiently simulate various types of conservation laws via *finite volume method* (FVM) discretization of the given PDE with the Euler explicit scheme [9]. The mathematical expression (*numerical scheme*) which has to be evaluated for each cell in each iteration can be represented with a synchronous data-flow graph. As the

DOI:10.15774/PPKE.ITK.2014.001

goal is to design a high-performance pipelined *arithmetic unit* (AU), which can operate at high frequency, each mathematical operation, i.e., node of the graph, is implemented with a dedicated *floating-point unit* (FPU). On recent high-end FPGAs, several floating-point units can be realized, which can operate at high frequency, however, the global control signals connected to each floating-point unit slow down the operating frequency of the rest of the circuit.

My research goal was to develop a novel design methodology which constructs high-performance, locally controlled AUs from synchronous data-flow graphs. My questions were the following: How shall I partition the data-flow graph to obtain clusters which can be controlled efficiently? How to control the clusters and how to connect them to avoid synchronization problems? What is the price of the improved frequency in terms of speed, power and area? Finally, how to automate the generation process of the AUs to drastically decrease the development time of new numerical simulations?

The second problem I investigated was the *Density Matrix Renormalization Group* (DMRG) algorithm [10]. The algorithm is a variational numerical approach, which has become one of the leading algorithms to study the low energy physics of strongly correlated systems exhibiting chain-like entanglement structure [11]. The algorithm was developed to balance the size of the effective Hilbert space and the accuracy of the simulation, and its runtime is dominated by the iterative diagonalization of the Hamilton operator. As the most time-consuming step of the algorithm, which is the projection operation of the diagonalization, can be expressed as a sequence of dense matrix operations, the DMRG is an appealing candidate to fully utilize the computing power residing in novel parallel architectures.

As the algorithm had not been accelerated on parallel architectures (to the best of my knowledge), my research goal was to investigate on which architecture the algorithm can be implemented most efficiently. My objective was to give a high-performance, parallel and flexible implementation on the selected architecture, which can deal with wide range of DMRG configurations.

In Chapter 2, modern parallel architectures are reviewed focusing on the architectures utilized in the dissertation: the FPGA and the GPU. In Chapter 3, an FPGA acceleration of the numerical solution of PDEs is presented including the numerical formulas of a Computational Fluid Dynamics (CFD) problem and the proposed FPGA framework. In Chapter 4, the proposed design methodology for constructing

DOI:10.15774/PPKE.ITK.2014.001

high-performance, locally controlled AUs is described. This chapter contains the scientific work related to my first thesis group. In Chapter 5, the DMRG algorithm is summarized including a run-time analysis of the CPU-only code and an estimation of the performance of a possible FPGA acceleration. In Chapter 6, a hybrid GPU-CPU DMRG code is described including the acceleration of the projection operation and some asymmetric matrix-vector operations of the diagonalization. This chapter contains the scientific work related to my second thesis group. Finally, the new scientific results of the dissertation are summarized in Chapter 7.

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

Chapter 2

Parallel architectures

Parallel architectures are being designed from the beginning of high performance computing, however, since clock frequency reached the physical limits, they have got into the primary focus of chip makers. At the current 20-30 nm technology, several processing elements (*cores*) can be packed into one chip and the resulting parallel architectures can be used to continue the performance improvement of the CMOS chips for some extra years. The demand for new parallel architectures raised several architectural questions which was answered differently by chip designers. Architectural differences are also due to the different application areas that different chip makers may target. Unfortunately, there is a very serious barrier in the way of packaging many cores, that is, many transistors into a single chip: the power wall. As the power consumption of a single transistor cannot be decreased below a physical limit, it is a constant challenge for chip designers to pack more performance into a single chip by designing more power efficient cores. (One possible way to decrease the power consumption of a chip is to "turn off" the parts which are currently not used, however, in high-performance applications, where all the resources are continuously utilized, this cannot be exploited.)

Multicore processors can be grouped into traditional multicore processors, in which a couple of heavy weight processing cores are glued on a single chip and to non-traditional multicore processors, which represent all the other efforts to design novel parallel architectures. The first group contains the traditional desktop and mobile CPUs of Intel and AMD, while the second group includes parallel efforts, such as IBM Cyclops-64 [12], IBM Cell [13], GPUs of NVidia and AMD, FPGAs of Xilinx and

DOI:10.15774/PPKE.ITK.2014.001

Altera, Intel Larrabee [14], Intel MIC [15], and Tilera Tile-Gx [16].

One of the main questions arising in the many-core architecture design is the question of memory hierarchies and caches. At the dawn of novel parallel architectures it was still an open question, how to supply program developers with expensive automated caching in complex systems. IBM chose a user-managed memory hierarchy approach in its Cell and Cyclops-64 processors. Cell processor was originally designed for Sony PlayStation 3 in the first half of 2000s, and it was reused in the IBM Roadrunner supercomputer which was the first computer breaking through the ”petaflop barrier”. Cyclops-64 processor was also developed during the first half of 2000s and was one of the first projects to pack dozens of cores into a single chip: it contained 80 cores reaching 80 GFlops performance. Despite the success they reached, the architectures have been discontinued in the second half of 2000s. One of the main bottlenecks of their widespread usage was that it was relatively hard to develop efficient programs on them (compared to rivals) due to the user-managed memory hierarchy.

Graphical processing units (GPUs) entered the competition of general purpose parallel architectures in 2006, when NVidia introduced its first unified shader architecture. The proposed architecture introduced an universal processing core (stream processor) instead of vertex and pixel shader processors. The NVidia Geforce 8800 processor (codename G80) consisted of 128 stream processors grouped into 8 blocks by 16 processors. Each stream processor was a very simple, but universal processing core which was capable of floating-point operations. From the aspect of memory hierarchy, they took a middle course by implementing both automated and non-automated caches. Nowadays, GPUs have an important role in high-performance computing, and the evolution of the architecture has not stopped. To review some of the new features of the modern GPUs, the NVidia Kepler architecture is presented in Section 2.2.

The success of GPUs inspired Intel to design its own GPU chip called Larrabee. The idea behind the new architecture was to combine the advantages of CPUs and GPUs. Larrabee was intended to use very simple, but x86 instruction set cores and automated cache mechanism across all the cores. The architecture was planned to contain 32 computing cores and to reach one TFlops performance in case of single precision. Although the chip had some working copies, it was not commercially released, and the architecture was replaced by the Intel Many Integrated Core (MIC) architecture. The Intel MIC architecture was debuted under the brand name Xeon Phi in 2012. Intel

DOI:10.15774/PPKE.ITK.2014.001

Xeon Phi is basically a coprocessor, which can be connected to an Intel Xeon host processor through a PCI express bus. The coprocessors of Intel Xeon Phi 7100 family contain 61 x86 instruction set cores connected via a very high bandwidth, bidirectional ring interconnect. Each core contains a vector processing unit with 512-bit SIMD instruction set, which can execute 32 single-precision or 16 double-precision floating point operations per cycle (assuming multiply-and-add operation). In contrast with the GPUs, all cache memories are fully coherent and implement the x86 memory order model. Each core is equipped with a 32 KB L1 instruction cache, a 32 KB L1 data cache, and a 512 KB unified L2 cache. In case of double precision, the theoretical peak performance of the coprocessor is 1208 GFlops, while the peak memory bandwidth is 352 GB/s.

In 2007, Tilera Corporation entered the high-performance multiprocessor industry with a novel many-core processors topology, in which the cores are interconnected via a 2D non-blocking mesh, called iMesh, forming an on-chip network. The Tile-Gx72 chip contains 72 processing cores, each of which is a full-featured 64-bit processor with fully coherent L1 and L2 caches. The chip integrates four DDR3 memory controllers (up to 1TB memory with 60 GB/s) and several I/O controllers (e.g. 8 10 Gb/s Ethernet, 6 PCI express ports), therefore, there is no need for a host CPU. The product targets the cloud computing industry by being 3-4 times more energy efficient than Intel's x86 based servers [17].

The FPGA represents a broader class of architectures as it is an universal chip which can be programmed to realize different types of architectures. However, as it can realize high-performance and power efficient parallel architectures, which compete with the previously mentioned chips, it can also be regarded as a highly customizable parallel architecture.

In the following sections, the FPGA and the GPU, the two architectures which are used in the dissertation, are presented in more detail.

2.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs), introduced by Xilinx Inc in 1985, are silicon devices that can be electrically programmed to realize wide range of digital circuits. They were created to fill the gap between fixed-function Application Specific

DOI:10.15774/PPKE.ITK.2014.001

Integrated Circuits (ASICs) and software programs. From development aspects, they are still between the two approaches and, sometimes, the only compromise for applications. In one hand, the fabrication time of ASICs cannot be compared to FPGAs that can be configured in less than a second. On the other hand, the development of an FPGA design usually takes significantly more time than a software development process. The reverse can be observed in case of power consumption: ASICs have the lowest power consumption, while FPGAs have lower power consumption than common CPUs running the software programs. The speed performance of an FPGA is typically 3-4 times smaller than an ASIC [18], and the maximal operating frequency of an FPGA is 5-6 times smaller than a high-end CPU. Comparing the overall performance of FPGA and CPU is problematic because their performance is task dependent and in specific computational domains the FPGA outperforms the CPU contrary to the slower operating frequency.

2.1.1 The general structure

The schematic structure of an FPGA is illustrated in Figure 2.1. The main building blocks are the generic logic blocks, the dedicated blocks (e.g. memory or multiplier), the routing fabric, and the I/O blocks. During the implementation of a circuit, which is designed for FPGA, the blocks and the routing resources are electrically configured (*programmed*). The technology relies on electrically programmable switches which select the proper operation in the configurable blocks or the interconnection topology in the routing fabric. Currently, most FPGAs use static memory (SRAM) switches, but special FPGAs are also built with flash or anti-fuse technology. The main advantage of SRAM switches is the fast operation, however, they have to be reconfigured each time they are powered on.

In the beginning, FPGAs contained only very simple logic blocks, however, nowadays, FPGAs contains complex logic blocks, which can be operated more efficiently in general. As a representative example, the logic block of Xilinx Virtex-6 FPGAs, also called Configurable Logic Block (CLB) is considered here. Each CLB contains two smaller logic cells, called slices. Every Virtex-6 slice contains four logic-function generators, also called look-up tables (LUTs), eight storage elements, wide-function multiplexers (MUX), and carry logic, which connects neighboring slices. To illustrate

DOI:10.15774/PPKE.ITK.2014.001

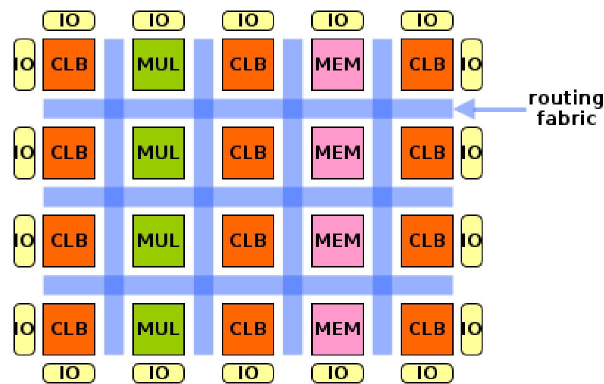

Figure 2.1: Schematic view of a simplified FPGA containing only a few building blocks: configurable logic (CLB), multiplier (MUL), memory (RAM), and I/O blocks.

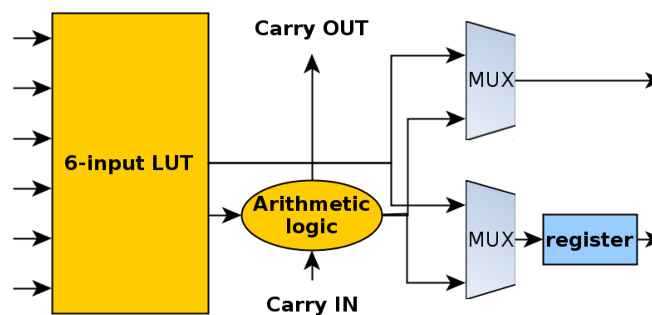

Figure 2.2: Schematic of a simplified logic cell

the concept, a simplified logic cell containing one 6-input LUT, two multiplexers, and one register is depicted in Figure 2.2. The 6-input LUT can be configured as a logic function, a 32-bit shift register, or a 64 bit memory. The storage element can act as a flip-flop or a latch. In case of Virtex-6, two types of slices are exist: SLICEL and SLICEM. In the simpler SLICELs, the LUTs cannot be configured to shift register or memory, consequently, they provide only logic and arithmetic functions. The more complex SLICEM, in addition to these functions, can be configured to store data using distributed RAM or to shift data with 32-bit registers.

The dedicated memory blocks are called block RAMs (BRAMs). They were introduced to utilize chip area more efficiently in memory hungry applications.They are usually placed in specific columns of the FPGA and can be used independently, or mul-

DOI:10.15774/PPKE.ITK.2014.001

tiple blocks can be combined together to implement larger memories. The blocks can be used for a variety of purposes, such as implementing standard single- or dual-port RAMs, first-in first-out (FIFO) functions, or state machines.

For some functions, like multipliers, which are frequently used in applications, it is worth to create dedicated blocks as well. If these functions are implemented via generic logic blocks connected together, the connections result in a significantly lower frequency than in case of ASIC. By introducing hardwired solutions for the most frequently used functions, the operating frequency can get one step closer to the frequency used in ASIC.

In the Virtex-6 family the dedicated multipliers blocks are called DSP48E1 blocks (here DSP stands for *digital signal processing*). These hardwired blocks are very flexible and provide several independent functions, such as multiply, multiply accumulate (MACC), multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter. Furthermore, the architecture supports cascading multiple DSP48E1s to form wider math functions, DSP filters, and complex arithmetic without the use of general FPGA logic.

The architecture also supports the so called *System-on-Chip*(SoC) solution at both software and hardware level. Since the number of transistors in a single chip reached a practical limit, complete systems can be realized on a single chip. At software level, several off-the-shelf softcore processors are available, what an FPGA developer can be include in the design if there are enough resources in the selected FPGA. At hardware level, Xilinx launched the Zynq Family, in which a Dual ARM Cortex-A9 processor is integrated into the FPGA chip itself.

The development flow of an FPGA-based design describes how an abstract textual Hardware Description Language (HDL) description of the design is converted to the device cell-level configuration. In the presented work, the VHDL language was used, and its name stands for Very High-Speed Integrated Circuit Hardware Description Language. The VHDL is a strongly typed, Ada-based programming language, which provides a structural and a register-transfer-level (RTL) descriptions for circuits. The structural description lets the programmer compose a design from sub-circuits, while, in RTL, the logic of sub-circuits can be described as transformations on data bits between register stages.

DOI:10.15774/PPKE.ITK.2014.001

The first step of a typical development process is the *testbench* creation, which can also be described in VHDL and is used to simulate the operation of the design at RTL level. If the design operates as desired, the *synthesis* process transforms the VHDL constructs to generic gate-level components, such as simple logic gates and registers. The next process is the *implementation* process, which consists of three smaller processes: *translate*, *map*, and *place-and-route*. First, the translate merges multiple design files into a single netlist. Next, the generic gates of the netlist are mapped to logic cells and I/O blocks. Finally, the place-and-route process places the cells into physical locations inside the FPGA chip and determines the routes for the connecting signals. At the end of the development flow, static timing analysis can be done to determine various timing parameters, such as maximal clock frequency, and the configuration file is *generated*.

2.1.2 The common peripherals

One of the main strengths of the FPGA is that it can support wide range of peripherals. In theory, the FPGA can be configured to handle any peripherals, however, in practice, we usually rely on the peripherals what FPGA manufacturers already support (e.g. via specific blocks or IP cores) and what FPGA board manufacturers place on the FPGA boards.

The I/O blocks of a Virtex-6 FPGA can be grouped into general purpose and transceiver I/O blocks.

2.1.2.1 General purpose I/Os

In Xilinx Virtex-6 FPGAs, general purpose I/O blocks are called Select I/O blocks and support a wide variety of standard interfaces. The I/O blocks are grouped into I/O banks, and blocks of the same bank usually implement the same standard, e.g. output buffers within the same bank must share the same output drive voltage. In a typical Virtex-6 FPGA, there are around 9-18 I/O banks and 360-1200 user-configurable I/O blocks.

Onboard DDR memories are usually connected to FPGAs through the general purpose I/O blocks. Their typical size is around 1-4 GB. In theory, larger memories could also be connected to an FPGA, however, in practice, FPGA board manufacturers do

DOI:10.15774/PPKE.ITK.2014.001

not pack more memories into the boards because it is unprofitable. Virtex-6 FPGAs support both DDR2 (800 Mb/s per pin) and DDR3 (1066 Mb/s per pin) interfaces. Assuming an Alpha Data ADM-XRC-6T1 FPGA board as a target research platform, the FPGA can be connected to 4 1GB memory modules. In this configuration, each memory module is 32bit wide and connected to a separate channel, consequently, the total memory throughput is 17.05 GB/s.

2.1.2.2 Transceiver I/Os

A transceiver is a combined transmitter and receiver for high-speed serial communication. In the investigated Virtex-6 family, two types of transceivers exist: GTX (6.6 Gb/s per lane) and GTH (11.18 Gb/s per lane). In the newer Virtex 7 family even a faster transceiver (GTZ) has been introduced with 28 Gb/s throughput per lane. These extremely fast transceivers make FPGA the only choice for some ultra-high bandwidth wired telecommunication applications.

The PCI express communication is also established through the transceivers. The PCI express throughput is very important in scientific applications because the CPU and the FPGA communicate via this interface. (Usually, the FPGA board is inserted into one of the PCI express slots of a host computer.) Virtex-6 family supports the first (2.5 Gb/s) and the second (5 Gb/s) generation of the protocol, while the newer Virtex 7 family supports the third generation (8 Gb/s) as well. Xilinx provides integrated interface blocks for PCI express designs, which are dedicated hard IPs to help the implementation of the protocol. Each block supports x1, x2, x4, or x8 lane configurations and Virtex-6 FPGAs contains usually 1-4 blocks. Assuming the previously mentioned Alpha Data board, the FPGA chip is connected to the PC via 4 lanes, which results in a modest 2 GB/s throughput. In theory, significantly faster configurations are also possible if more blocks and more lanes are utilized.

Transceivers enable communication through other interfaces as well, such as Serial Advanced Technology Attachment (SATA) or Ethernet. All revisions of SATA interface are supported by transceivers enabling 6 Gb/s throughput, however, some layers of the protocol have to be implemented in the general FPGA logic via a soft IP core. Ethernet communication is supported by a dedicated block called Tri-mode Ethernet MAC (TEMAC) to save logic resources and design efforts. The maximal communication throughput supported by the Virtex-6 family is 2.5 Gb/s.

DOI:10.15774/PPKE.ITK.2014.001

2.1.3 Xilinx Virtex-6 SX475T FPGA

The Xilinx Virtex-6 SX475T FPGA with speed grade -1, which has been used in the dissertation, is part of the Virtex-6 SXT subfamily, which was designed to deliver the highest ratio of DSP and memory resources for high-performance applications. The most important parameters of the Virtex-6 familiy are summarized in Table 2.1. The chip contains 74400 slices, 2016 multiplier blocks (DSP48E1), 38304 Kb on-chip memory (BRAM), 2 interface blocks for PCI express, 4 Ethernet MACs, 840 general purpose I/O blocks and 36 GTX transceivers.

To estimate the computational performance of the selected FPGA, we can assume a heavily pipelined architecture that is not memory bandwidth limited, e.g. dense matrix-matrix multiplication. In this type of applications, the performance is only limited by the number of operation units that can be implemented. To estimate the fixed point performance, the Xilinx LogiCORE IP Multiplier soft IP can be assumed as the basic operation unit. If it is configured to a fixed point 25x18 multiplier, it can be operated at 450 MHz and occupies 1 DSP48E1 blocks. In this configuration, 2016 multipliers can be implemented, and each can start a combined multiply-and-add operation (MAD) for every clock cycle resulting in 1814.4 giga operations per second. To estimate the floating-point performance, the Xilinx LogiCORE IP Floating-Point Operator can be used. In single precision case, it can operate at 429 MHz and occupies 3 DSP blocks. In the selected FPGA, 672 multiplication units can be implemented and the remaining logic still allows 509 adder units, which can be connected to the multipliers resulting in 436.7 GFlops performance. In double precision case, the multiplication can operate at 429 MHz and occupies 11 DSP blocks. There are enough resources for 183 multiplication and adder units, however, the adder unit can operate at only 361 MHz resulting in 132.1 GFlops performance.

2.2 Graphical Processing Units

After the introduction of the unified shader architecture, the fast development of the GPU architecture has been continued. The next move toward general-purpose computing happened in late 2007, when AMD introduced the double-precision support in Radeon HD 3800 series and FireStream 9170. In 2008, the double-precision support

DOI:10.15774/PPKE.ITK.2014.001

Table 2.1: Virtex-6 FPGA Feature Summary

Device	Configurable Logic Blocks (CLBs)		DSP48E1 Slices	Block RAM Blocks (36Kb)	Maximum Transceivers GTX	Max User I/O
	Slices	Max Distributed RAM (Kb)				
XC6VLX75T	11,640	1,045	288	156	12	360
XC6VLX130T	20,000	1,740	480	264	20	600
XC6VLX195T	31,200	3,040	640	344	20	600
XC6VLX240T	37,680	3,650	768	416	24	720
XC6VLX365T	56,880	4,130	576	416	24	720
XC6VLX550T	85,920	6,200	864	632	36	1200
XC6VLX760	118,560	8,280	864	720	0	1200
XC6VSX315T	49,200	5,090	1,344	704	24	720
XC6VSX475T	74,400	7,640	2,016	1,064	36	840
XC6VHX250T	39,360	3,040	576	504	48	320
XC6VHX255T	39,600	3,050	576	516	24	480
XC6VHX380T	59,760	4,570	864	768	48	720
XC6VHX565T	88,560	6,370	864	912	48	720

also appeared at NVidia in the second generation of the unified shader architecture, called Tesla architecture. Traditionally, the GPU cards containing the GPU chip were connected to the host system via a PCI express slot and to the monitor via a video display interface, however, starting from these new architectures, both GPU manufacturers started to offer special cards without video display for general-purpose high-performance computing.

For programming GPUs, both manufacturers introduced their own development platforms: CUDA SDK by NVidia and Stream SDK by AMD. In 2008, OpenCL, a general framework for writing programs which can be executed on various parallel architectures, was introduced. AMD decided to support OpenCL instead of its previous framework, while NVidia decided to support both OpenCL and CUDA. Although the OpenCL framework has the theoretical advantage that it can create universal program code, the performance of the universal code is sometimes far behind the native solution. Usually, even in OpenCL, some hardware specific adjustments have to be done in the code to create a high-performance implementation, which leads to the following questions: Is it possible to describe wide range of architectures without a significant loss in the performance? Will OpenCL compilers be smart enough to compete with the native SDKs?

In the dissertation the NVidia K20 GPU has been selected for the demonstration of

DOI:10.15774/PPKE.ITK.2014.001

the implemented DMRG program. The selected GPU is one of the leading GPUs providing more than one teraflops performance in double-precision and can be regarded as an illustrative example representing the modern GPU architecture. In the following section, the NVidia Kepler architecture, on which the K20 is based, is presented in more detail. As the DMRG program was implemented via CUDA, the CUDA SDK is also introduced.

2.2.1 NVidia Kepler architecture

The Kepler architecture was released in 2012 and it can be regarded as the 4th major improvement of the GPU architecture since the introduction of the unified shader architecture. Compared to the previous generation (called Fermi), the manufacturer primarily focused on the minimization of the power consumption, and in the high-performance GK110 chip the double-precision performance was also increased. The main parameters of the NVIDIA Tesla products built upon the Kepler or the Fermi architectures are summarized in Table 2.2. The new architecture was reported 3 times more power efficient than the Fermi architecture [19] and introduced several new features, such as Dynamic Parallelism or NVidia GPUDirect. Dynamic Parallelism enables the programer to write smarter kernels which can dispatch new kernels without host intervention. NVidia GPUDirect is a new communication way, in which the GPU memory can be directly accessed via the PCI express interface eliminating CPU bandwidth and latency bottlenecks. The GPU can be directly connected to a *network interface controller* (NIC) to exchange data with other GPUs via *Remote Direct Memory Access* (RMDA). The GPU can also be connected to other 3rd party devices, e.g. storage devices.

2.2.1.1 The general structure

A schematic block diagram of the Kepler GK100 chip is displayed in Figure 2.3. The chip is associated with CUDA Compute Capability 3.5, which is the revision number of the underlying architecture and determines the available CUDA features. The architecture contains 15 Streaming Multiprocessors (SMX) and 6 64bit memory controllers. Each SMX contains 192 single-precision CUDA cores, 64 double-precision units, 32 special function units, 65356 32bit registers, 64 KB shared memory, 48 KB read-only

DOI:10.15774/PPKE.ITK.2014.001

Table 2.2: NVIDIA Tesla product line and the GTX 570 GPU, which was also tested in the dissertation.

	GTX 570	M2050	M2070/ M2075	M2090	K10	K20	K20X
Architecture	GF110 (Fermi)				2xGK104 (Kepler)	GK110 (Kepler)	
Number of CUDA cores	448	448	448	512	2x1536	2496	2688
Core clock frequency (MHz)	1464	1150	1150	1300	745	706	732
Onboard memory size (GB)	1.3	3	6	6	8	5	6
Onboard memory bandwidth (GB/s)	152	148	150	177	320	208	250
Peak double precision floating point performance (GFlops)	175	515	515	665	190	1170	1310
Peak single precision floating point performance (GFlops)	1405	1030	1030	1331	4580	3520	3950
Compute capability	2.0	2.0	2.0	2.0	3.0	3.5	3.5

data cache, and 4 warp schedulers. SMX supports the IEEE 754-2008 standard for single- and double-precision floating-point operations (e.g. fused multiply-add) and can execute 192 single-precision or 64 double-precision operations per cycle. Special function units can be used for approximate transcendental functions such as trigonometric functions.

2.2.1.2 CUDA programming

The CUDA SDK, which was debuted in 2006, is a general computing framework and a programing model that enables developers to program the CUDA capable devices of NVidia. Programs running on CUDA capable devices are called *kernels*. Kernels are programmed in CUDA C, which is a standard C with some extensions. Kernels can be dispatched from various supported high-level programming languages such as C/C++ or Fortran, and there are also CUDA libraries, which collect kernels written for specific applications (e.g. CuBLAS for Basic Linear Algebra Subroutines).

When a kernel is dispatched, several threads are started to execute the same kernel code on different input. The mechanism is called Single Instruction Multiple Threads (SIMT), and one of the key differences from Single Instruction Multiple Data (SIMD)

DOI:10.15774/PPKE.ITK.2014.001

Figure 2.3: A schematic block diagram of the Kepler GK110 chip. Image source: NVidia Kepler Whitepaper [19]. The chip contains 15 Streaming Multiprocessors (SMX). Cache memories, single-precision CUDA cores, and double-precision units are indicated by blue, green, and orange, respectively.

DOI:10.15774/PPKE.ITK.2014.001

is that threads can access input data via an arbitrary input pattern. Threads are organized into a *thread hierarchy*, which is an important concept in CUDA programming. The programmer can determine the number and the topology (1D, 2D, or 3D) of threads to form a *thread block* and several thread blocks can be defined to form a grid. The total number of threads shall match to the size of the problem that the treads have to solve. During execution the tread blocks are distributed among the available stream processors.

The second important concept in CUDA programming is the *memory hierarchy*. At thread level, each thread can use private (local) *registers* allocated in the register file of SMX, which is the fastest memory. At thread block level, threads of a block can reach a *shared memory* allocated in the shared memory of SMX. Finally, at grid level, all threads can access the on-board DDR memory, which is the largest but slowest memory on the GPU card.

When thread blocks are assigned to an SMX, the threads of the assigned blocks can be executed concurrently; even execution of threads of different blocks can be overlapped. The SMX handles the threads in groups of 32, called *warps*. The SMX distributes the warps between its four warp schedulers, and each sheduler schedules the execution of the assigned warps to hide various latencies. Each scheduler can issue two independent *instructions* for one of its warps per clock cycles, that is, an SMX can issue eight instructions per clock cycle if the required instruction level parallelism and functional units are available. As a warp contains 32 threads, one instruction corresponds to 32 operations. To give an example, 6 single-precision instructions can be executed on the 192 CUDA cores and 2 double-precision instructions can be executed on the 64 double-precision computing units. As there is no branch prediction, all threads of a warp shall agree on their execution path for maximal performance.

2.2.1.3 NVidia K20

The NVIDIA Tesla K20 graphics processing unit (GPU), which is utilized in the dissertation, is an extension board containing a single GK110 chip. The board is connected to the host system via an x16 PCI Express Generation 2 interface, which provides 8 GB/s communication bandwidth. The board contains 5 GB GDDR5 memory, which is

DOI:10.15774/PPKE.ITK.2014.001

accessed through a 320-bit interface providing 208 GB/s memory throughput. The estimated power consumption of the full board during operation is approximately 225 W.

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

Chapter 3

Solving Partial Differential Equations on FPGA

3.1 Computational Fluid Dynamics (CFD)

A wide range of industrial processes and scientific phenomena involve gas or fluid flows over complex obstacles, e.g. air flow around vehicles and buildings, the flow of water in the oceans or liquid in BioMEMS. In such applications, the temporal evolution of non-ideal, compressible fluids is often modeled by the system of Navier-Stokes equations. The model is based on the fundamental laws of mass-, momentum- and energy conservation, including the dissipative effects of viscosity, diffusion and heat conduction. By neglecting all non-ideal processes and assuming adiabatic variations, we obtain the Euler equations.

3.1.1 Euler equations

The Euler equations [20, 21], which describe the dynamics of dissipation-free, inviscid, compressible fluids, are a coupled set of nonlinear hyperbolic partial differential equations. In conservative form they are expressed as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{3.1a}$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot \left(\rho \mathbf{v}\mathbf{v} + \hat{I} p\right) = 0 \tag{3.1b}$$

DOI:10.15774/PPKE.ITK.2014.001

$$\frac{\partial E}{\partial t} + \nabla \cdot ((E + p)\mathbf{v}) = 0 \tag{3.1c}$$

where t denotes time, ∇ is the Nabla operator, ρ is the density, u, v are the x- and y-component of the velocity vector $\mathbf{v}$, respectively, p is the thermal pressure of the fluid, $\hat{I}$ is the identity matrix, and E is the total energy density defined by

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho\mathbf{v} \cdot \mathbf{v}. \tag{3.1d}$$

In equation (3.1d) the value of the ratio of specific heats is taken to be $\gamma = 1.4$.

The quantities $\rho, \rho\mathbf{v}, E$ can be grouped into the conservative state vector $\mathbf{U} = [\rho, \rho\mathbf{v}, E]^T$, and it is also convenient to merge (3.1a), (3.1b) and (3.1c) into hyperbolic conservation law form in terms of $\mathbf{U}$ and the flux tensor

$$\mathbf{F} = \begin{pmatrix} \rho\mathbf{v} \\ \rho\mathbf{v}\mathbf{v} + \hat{I}p \\ (E + p)\mathbf{v} \end{pmatrix} \tag{3.2}$$

as:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F} = 0. \tag{3.3}$$

The equation expresses the conservation of quantity $\mathbf{U}$: the change of $\mathbf{U}$ equals to the divergence of $\mathbf{F}$ which describes the transport mechanism of $\mathbf{U}$.

3.1.2 Finite volume method solution of Euler equations

Finite Volume Method (FVM) is a frequently used discretization strategy to numerically solve the hyperbolic conservation laws expressed by the Euler equations [22]. To illustrate a possible application, a simulation of the airflow inside a scramjet engine is presented in Figure 3.1. During the discretization, a mesh is introduced for the *computational domain* where the PDE is studied. The idea behind the strategy is to integrate Equation 3.3 for each element (*cell*) of the mesh. Using the divergence theorem, the change of the quantity U inside each cell can be approximated via the fluxes going through the border of the cell:

$$\int_{\mathcal{T}} \frac{\partial \mathbf{U}}{\partial t} + \oint_{\mathcal{T}} \mathbf{F} \cdot \mathbf{n}_{\mathcal{T}} = 0. \tag{3.4}$$

DOI:10.15774/PPKE.ITK.2014.001

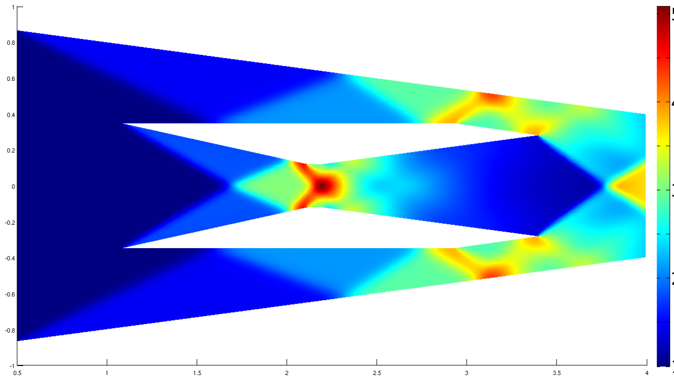

Figure 3.1: A simulation of the airflow inside a scramjet engine. Colors represent the value of the density (kg/m^3) component of the state vector.

where $\mathbf{n}_{\mathcal{T}}$ is the outward pointing unit normal field of the boundary of cell $\mathcal{T}$. As numerical fluxes are locally conserved, flux computations can be reused in the neighboring cells.

The temporal derivative is discretized by the first-order forward Euler method:

$$\frac{\partial \mathbf{U}}{\partial t} = \frac{\mathbf{U^{n+1}} - \mathbf{U^{n}}}{\Delta t}, \tag{3.5}$$

where $\mathbf{U^n}$ is the known value of the state vector at time level n, $\mathbf{U^{n+1}}$ is the unknown value of the state vector at time level $n+1$, and Δt is the time step. In the current implementation, constant time steps are used, however, a variable time step modification of the framework is also possible.

In Chapter 4 the developed design methodology is presented via the solution of the Euler equations in case of structured and unstructured space discretization. In both cases the fluxes are computed similarly, however, in the unstructured case, extra rotations are needed which makes the numerical scheme more complex. In the following part both schemes are presented.

3.1.2.1 Structured mesh

Kocsardi et al. [23] showed that logically structured arrangement of data is a convenient choice for an efficient FPGA-based implementation. Hereby I am shortly reviewing the

DOI:10.15774/PPKE.ITK.2014.001

numerical scheme because it is used as a test case to demonstrate the AU generation capabilities of the presented framework.

The computational domain Ω is composed of $n \times m$ logically structured rectangles (cells). The area of a cell $\mathcal{T}$ is denoted by $V_{\mathcal{T}}$, while a face f is described by the vector $\mathbf{n_f}$ which is normal to the face f and its size equals to the area of the face. The flux tensor evaluated at a face f is denoted by $\mathbf{F_f}$. Using the cell centered FVM discretization, the governing equations (Equation 3.4) can be described as

$$\frac{\partial \mathbf{U}_{\mathcal{T}}}{\partial t} = -\frac{1}{V_{\mathcal{T}}} \sum_f \mathbf{F_f} \cdot \mathbf{n_f}, \tag{3.6}$$

where the summation is meant for all four faces of cell $\mathcal{T}$.

In order to stabilize the solution procedure, artificial dissipation has to be introduced into the scheme. According to the standard procedure, this is achieved by replacing the physical flux tensor by a numerical flux function $\tilde{\mathbf{F}}$ containing a dissipative stabilization term. In the dissertation the simple and robust Lax-Friedrichs numerical flux function is used, which is defined as

$$\tilde{\mathbf{F}} = \frac{\mathbf{F_L} + \mathbf{F_R}}{2} - (|\bar{u}| + \bar{c}) \frac{\mathbf{U_R} - \mathbf{U_L}}{2} \tag{3.7}$$

where $\mathbf{F_L}(\mathbf{U_L})$ and $\mathbf{F_R}(\mathbf{U_R})$ is the flux at the left and right side of the interface, respectively, $|\bar{u}|$ is the average value of the velocity, and $|\bar{c}|$ is the speed of sound at the interface.

Fixing the coordinate system for each cell in an east-south direction, the dot product with the normal vector can be simplified to a multiplication by the area of the face. In case of eastward and southward fluxes the first and second columns of the flux tensor can be used, respectively:

$$\tilde{F}^{\rho}_{east} = \frac{\rho_L u_L + \rho_R u_R}{2} - (|\bar{u}| + \bar{c}) \frac{\rho_R - \rho_L}{2} \tag{3.8a}$$

$$\tilde{F}^{\rho u}_{east} = \frac{(\rho_L u_L^2 + p_L) + (\rho_R u_R^2 + p_R)}{2} - (|\bar{u}| + \bar{c}) \frac{\rho_R u_R - \rho_L u_L}{2} \tag{3.8b}$$

$$\tilde{F}^{\rho v}_{east} = \frac{\rho_L u_L v_L + \rho_R u_R v_R}{2} - (|\bar{u}| + \bar{c}) \frac{\rho_R v_R - \rho_L v_L}{2} \tag{3.8c}$$

$$\tilde{F}^{E}_{east} = \frac{(E_L + p_L)\, u_L + (E_R + p_R)\, u_R}{2} - (|\bar{u}| + \bar{c}) \frac{E_R - E_L}{2} \tag{3.8d}$$

DOI:10.15774/PPKE.ITK.2014.001

$$\tilde{F}^{\rho}_{south} = \frac{\rho_L v_L + \rho_R v_R}{2} - (|\bar{u}| + \bar{c})\,\frac{\rho_R - \rho_L}{2} \tag{3.9a}$$

$$\tilde{F}^{\rho u}_{south} = \frac{\rho_L u_L v_L + \rho_R u_R v_R}{2} - (|\bar{u}| + \bar{c})\,\frac{\rho_R u_R - \rho_L u_L}{2} \tag{3.9b}$$

$$\tilde{F}^{\rho v}_{south} = \frac{(\rho_L v_L^2 + p_L) + (\rho_R v_R^2 + p_R)}{2} - (|\bar{u}| + \bar{c})\,\frac{\rho_R v_R - \rho_L v_L}{2} \tag{3.9c}$$

$$\tilde{F}^{E}_{south} = \frac{(E_L + p_L)\,v_L + (E_R + p_R)\,v_R}{2} - (|\bar{u}| + \bar{c})\,\frac{E_R - E_L}{2} \tag{3.9d}$$

where L indicates the cell where the state vector is updated, while R indicates the neighboring cell (e.g. east or south). In case of northward and westward fluxes, the flux has already been computed at one of the neighboring cells and can be reused with a minus sign. Applying time discretization and using the numerical fluxes the state vectors can be updated according to the following formula:

$$\mathbf{U}_{\mathcal{T}}^{\mathbf{n+1}} = \mathbf{U}_{\mathcal{T}}^{\mathbf{n}} - \frac{\Delta t}{V_{\mathcal{T}}} \sum_{f} \tilde{\mathbf{F}}_{\mathbf{f}} |\mathbf{n}_{\mathbf{f}}|, \tag{3.10}$$

3.1.2.2 Unstructured mesh

Structured data representation is not flexible for the spatial discretization of complex geometries. One of the main innovative contributions of our paper [1] was that an unstructured, cell-centered representation of physical quantities was implemented on FPGA. In the following paragraphs the mesh geometry, the governing equations, and the main features of the numerical algorithm are presented.

The computational domain Ω is composed of non-overlapping triangles. Each face f of a triangle $\mathcal{T}$ is associated with a normal vector $\mathbf{n_f}$ which points outward $\mathcal{T}$ and is scaled by the length of the face. The volume of a triangle $\mathcal{T}$ is indicated by $V_{\mathcal{T}}$. Following the finite volume methodology, the state vectors are stored at the mass center of the triangles.

During flux computation the coordinate system is attached to the given face f such a way that axis x is normal to f (see Fig. 3.2). The benefit of this representation is that the $\mathbf{F_f} \cdot \mathbf{n_f}$ dot product equals to the first column of $\mathbf{F}$ multiplied by the area of the face. The drawbacks of the representation are that the velocity vectors have to be rotated to this coordinate system before the flux computation and fluxes have to be rotated back to the global coordinate system before final summation.

DOI:10.15774/PPKE.ITK.2014.001

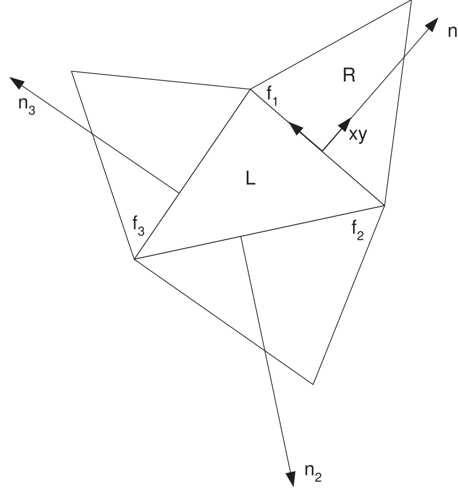

Figure 3.2: Interface with the normal vector and the cells required in the computation

Similarly to the structured case, the Lax-Friedrichs numerical flux function (Equation 3.7) is used to stabilize the solution, however, in this case, the normal component of the numerical flux function has to be calculated for each interface:

$$\tilde{F}_f^{\rho} = \frac{\rho_L u_L + \rho_R u_R}{2} - (|\bar{u}| + \bar{c})\,\frac{\rho_R - \rho_L}{2} \tag{3.11a}$$

$$\tilde{F}_f^{\rho u} = \frac{(\rho_L u_L^2 + p_L) + (\rho_R u_R^2 + p_R)}{2} - (|\bar{u}| + \bar{c})\,\frac{\rho_R u_R - \rho_L u_L}{2} \tag{3.11b}$$

$$\tilde{F}_f^{\rho v} = \frac{\rho_L u_L v_L + \rho_R u_R v_R}{2} - (|\bar{u}| + \bar{c})\,\frac{\rho_R v_R - \rho_L v_L}{2} \tag{3.11c}$$

$$\tilde{F}_f^{E} = \frac{(E_L + p_L)\,u_L + (E_R + p_R)\,u_R}{2} - (|\bar{u}| + \bar{c})\,\frac{E_R - E_L}{2} \tag{3.11d}$$

where L indicates the cell where the state vector is updated and R indicates the neighboring cell.

Applying time discretization and using the numerical fluxes, the state vectors can be updated according to the following formula:

$$\mathbf{U}_{\mathcal{J}}^{\mathbf{n+1}} = \mathbf{U}_{\mathcal{J}}^{\mathbf{n}} - \frac{\Delta t}{V_{\mathcal{J}}} \sum_f \hat{\mathcal{R}}_{\mathbf{n_f}} \tilde{\mathbf{F}}_{\mathbf{f}} |\mathbf{n_f}|, \tag{3.12}$$

where $\hat{\mathcal{R}}_{\mathbf{n_f}}$ is the rotation tensor describing the transformation from the face-attached coordinate system to the global one. Unfortunately, in the unstructured case, it is not

DOI:10.15774/PPKE.ITK.2014.001

worth to reuse the already computed neighboring fluxes because it doubles the on-chip memory requirement of the implementation.

The AU is generated from Equations 3.11a to 3.11d and is designed to compute the normal component of the numerical flux function for a new interface in each clock cycle. Beside the AU an additional simple arithmetic unit is required to update conservative state variables ($\rho, \rho u, \rho v, E$) using the flux vectors. As a cell has three interfaces, three clock cycles are required for a complete cell update.

3.2 Data structures and memory access patterns

The data structures used in the presented architecture were designed to efficiently use the available memory bandwidth during transmission of the unstructured mesh data to the FPGA. In numerical simulations data is discretized over space and can be represented at the vertices of the mesh (*vertex centered*) or the spatial domain can be partitioned into discrete cells (e.g. triangles) and the data is represented at the center of the cells (*cell centered*). From the aspect of accelerator design both vertex and cell centered discretization can be handled with a very similar memory data structure and accelerator architecture. In both cases the data can be divided into a time dependent (state variables) and a time independent part which contains mesh related descriptors (e.g. connectivity descriptor) and physical constants.

In the solution of the CFD problem which is presented in the dissertation, the cell centered approach is used, therefore the proposed data structures are explained for this case. In Figure 3.3 an example of an unstructured mesh is shown, in which the cells are ordered, and the computation of the new state values is carried out in an increasing order. Already processed cells are indicated by underlined numbers, the currently processed cell is encircled and squared cells are currently stored in the on-chip memory. On the right side of the figure, fragments of the appropriate data structures are illustrated.

Time dependent variables are composed of the state variables associated to the cells. When a cell is processed, the state variables are updated based on the state variables of the given cell and its neighborhood (*discretization stencil*). State variables are transmitted to FPGA in processing order and can be stored in a fixed-size shift register (on-chip memory). When a cell is processed, all the state variables of the cells from the

DOI:10.15774/PPKE.ITK.2014.001

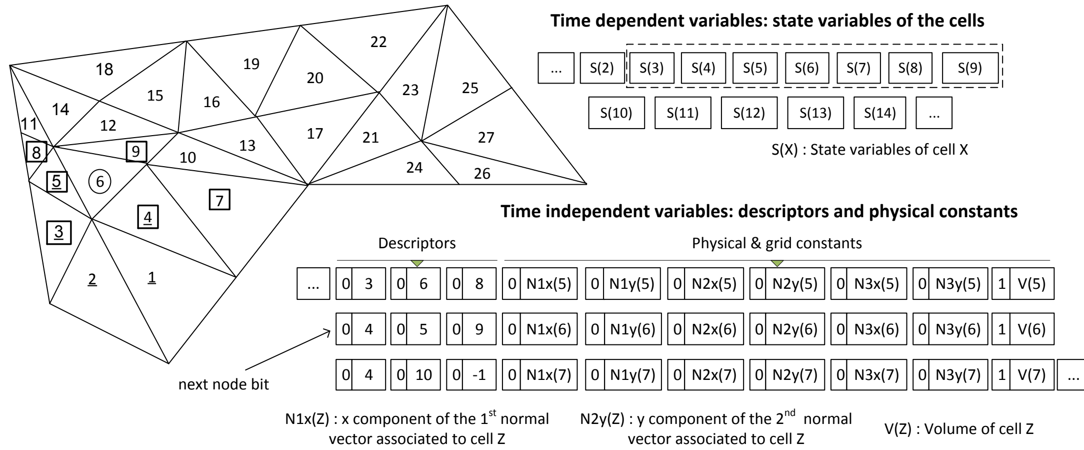

Figure 3.3: On the left, an unstructured mesh is shown to illustrate which cells are stored in the on-chip memory. Indices of the cells indicate the order of computation. Already processed cells are indicated by underlined numbers, the currently processed cell is encircled and squared cells are currently stored in the on-chip memory. On the right, fragments of the appropriate data structures are presented.

neighborhood stencil must be loaded into the on-chip memory, however, cells which have no unprocessed neighbors can be flushed out. In the presented example, when cell 6 is processed, all the necessary state variables are in the fast on-chip memory of the FPGA. To process the next cell (7), a new cell (10) should be loaded, and cell 3 can be discarded from the on-chip memory. It is possible that multiple new cells are required for the update of a cell, indicating that the on-chip memory is undersized. The size of the required on-chip memory depends on the structure of the grid and the numbering of the cells, consequently, a great attention should be paid for the ordering of the mesh points in a practical implementation. In the presented cell centered example the neighborhood stencil is relatively simple, however, more complicated patterns can be handled in the same way.

Time independent variables are composed of mesh related descriptors and other physical constants which are only used for the computation of the currently processed cell. They mainly differ from the time dependent variables because they are stored only for the currently processed cell and they are not written back to the off-chip memory. Consequently, time dependent and time independent data must be stored separately in

DOI:10.15774/PPKE.ITK.2014.001

the off-chip memory, otherwise the updated state variables cannot be transferred back to the off-chip memory in bursts.

Mesh related descriptors describe the local neighborhood of the cell (vertex) which is currently processed. In unstructured grids, the topology of the cells (vertices) can be described by a sparse adjacency matrix, which is usually stored in a Compressed Row Storage (CRS) format [24]. As the matrix is sparse and the vertices are read in a serial sequence (row-wise), the nonzero elements can be indicated by column indices and a *next node bit*, which indicates the start of a new row (see Figure 3.3). In the 2D cell centered case, the descriptor of a cell is a list, which contains the indices of the neighboring cells, however, in others cases (e.g. vertex centered) additional element descriptor may be required. Boundary conditions can be encoded by negative indices in the connectivity descriptor as illustrated in case of cell 7. If the size of the descriptor list is constant, the next node bit can be neglected.

Time independent data also contain physical constants which are needed for the computation of the new state values. These constants can be appended after the descriptors as shown in Figure 3.3. In case of the demonstrated CFD problem these constants are the normal vectors indicating the edges of the cells (triangle) and the volume of the cells.

3.3 Structure of the proposed processor

The processor were designed to efficiently operate on the input stream of the data of the ordered mesh points. The two main parts of the processor are the Memory unit and the Arithmetic unit (**AU**) as shown in Figure 3.4. The job of the Memory unit is to prepare the input stream and generate the necessary inputs to the AU, which should continuously operate and get new inputs in each clock cycle. The Memory unit could be reused with little adjustments if an other PDE needed to be solved, however, the AU would have to be completely reimplemented according to the new state equations. Consequently, the presented automatic generation and optimization of the AU can drastically decrease the implementation costs of a new problem.

The Memory unit is built from dual ported BRAM memories and stores the state variables of the relevant mesh points. The minimal size of the on-chip memory is determined by the bandwidth of the adjacency matrix of the mesh. The bandwidth

DOI:10.15774/PPKE.ITK.2014.001

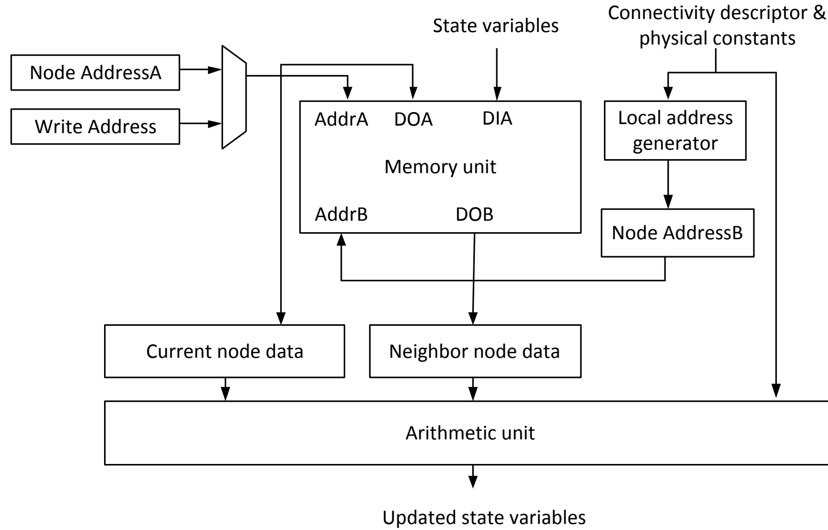

Figure 3.4: Block diagram of the proposed processor

of a matrix is defined as the maximum distance of a nonzero element from the main diagonal. The visiting order of the mesh points, that is, the rows and columns of the adjacency matrix can be reordered to minimize the bandwidth via several methods such as the widely used GPS algorithm [25] or the Amoeba1 algorithm presented in [1]. Offline reordering of the mesh points is a key step of the FPGA acceleration, because if the Memory unit is too large for the FPGA chip, the proposed architecture cannot be used.

In the unstructured CFD problem, each cell (triangle) has three interfaces, and the state variables are updated based on a flux function computed at the three interfaces. In each clock cycle, one interface of the given cell is evaluated by the AU, therefore, the new state variables can be computed in 3 or 4 cycles in case of 2D or 3D cell centered discretization, respectively. In the vertex centered discretization, the length of the computation is determined by the degree of the given vertex.

Computation is started by loading serial sequence of state variables into the Memory unit until it is half filled. Global indices of the neighboring cells are translated into addresses in the Memory unit by the Local address generator. In this phase the state variables of the first cell are loaded into the Current node register and the Neighborhood node register is filled by the state variables of the first neighbor using the

DOI:10.15774/PPKE.ITK.2014.001

incoming connectivity descriptor. When all neighbors of the first cell are sent to the AU, the *Node AddressA* register is incremented and the state variables of the second cell are loaded into the Current node register. During the next clock cycle, the state variables of a new cell data can be written into the Memory unit and computation of the second cell can be started. After an initial latency of the arithmetic unit, the updated state variables are written back to the off-chip memory in the same sequential order as they were loaded. The Memory unit is operating as a circular buffer; when it is filled the oldest cell data is overwritten. This can be safely done because the size of the memory is set to twice the bandwidth of the adjacency matrix, and the oldest values will not be required during the update of the remaining cells.

Descriptors add an overhead for the off-chip memory requirements and increase the memory bandwidth requirement of the processor. As the global index of the cells are never required and the order of cell data are statically scheduled, the memory address translation can be done offline. In this case the shorter local addresses can be stored and transferred, which significantly decreases the memory bandwidth requirement of the processor.

3.4 Outline of the multi-processor architecture

The high-level block diagram of the proposed architecture is shown in Figure 3.5. The memory interface provides the physical interface for the off-chip memory and the arbitration between the DMA engines competing for the memory. The sequential off-chip memory access pattern is a great advantage of the architecture because the off-chip memory can be accessed with optimal burst length, and the penalties of random access patterns can be eliminated. The DMA engines load the time dependent (states) and time independent (descriptors and constants) data into the corresponding input FIFOs of the processor in long sequential bursts, and the computed new state values are also written back to the off-chip memory in long sequential bursts.

The state of the system is usually saved after computing hundreds of explicit time steps during the computation, therefore the results of the first iteration can be fed directly into a second processor which computes the second iteration. The second processor must wait until its Memory unit is half filled to start computation. The results of the second iteration can be either saved into the off-chip memory or fed into another

DOI:10.15774/PPKE.ITK.2014.001

Figure 3.5: Outline of the proposed architecture. The processors are connected to each other in a chain to provide linear speedup without increasing memory bandwidth requirements. The number of processors is only limited by the available resources on the given FPGA.

processor. As the time independent data do not change between the iterations, an additional FIFO is enough to store the previously loaded descriptors and constants between the two computation stages. The depth of the FIFO is determined by the size of the Memory unit and the length of descriptors. The number of implemented processors is only limited by the available logic and memory resources on the FPGA. After a short initial startup latency, which is negligible compared to the number of mesh cells, the pipelined chain of processors can work in parallel providing linear speedup without increasing memory bandwidth requirements.

3.5 Analysis of the chosen design strategies

To design a high-performance competitive accelerator, the special properties of FPGAs have to be considered. In general, FPGAs are connected to a host PC through a relatively narrow PCI express interface, the DDR3 memories of FPGA boards are significantly slower than the GDDR5 memories of GPU boards, and also the operating frequency of FPGA circuits is significantly lower than the typical CPU/GPU frequencies. Due to these limitations, an FPGA acceleration of a computationally intensive problem can outperform the CPU/GPU implementations only if it reaches significantly better hardware utilization than the CPU/GPU implementations. As a consequence, for FPGA accelerators, the streaming architectures are advantageous, in which a custom, problem specific arithmetic unit can be designed, which utilizes all the FPGA resources

DOI:10.15774/PPKE.ITK.2014.001

and continuously processes the input data streamed into the FPGA chip.

On the one hand, the relatively narrow PCI express connection was not a limiting factor in my case, as the results of each iteration were not required to be transferred back to the host memory. On the other hand, the relatively slow onboard memories affected the chosen design strategies and an architecture with high computation per communication ratio had to be planned. These observations led to the presented solution, in which a sufficiently large onchip cache memory is assumed and the grid elements are visited in a specific order requiring each element to be loaded only once per iteration. As there are plenty of grid elements to be processed in each iteration, an arithmetic unit with a longer latency can be tolerated. As long as the accelerator is not memory bandwidth limited, its performance is determined by the operating frequency of the arithmetic unit. As a consequence, a pipelined arithmetic unit shall be implemented, in which higher operating frequency can be reached in exchange for a modest increase of the area and the latency.

In the proposed multi-processor configuration, the speed of the onboard memories is the reason why the processors cannot process more than one grid element in the same iteration. The memory throughput is not enough to supply more than one processor directly with grid element data. (On GPU, where the memory throughput is an order of magnitude larger, multiple processing elements can work on computations which belong to the same iteration.) In case of FPGA, if there are enough resources to implement more than one processor, they shall be connected after each other and work on computations associated to consecutive iterations. In this configuration, the extra processors do not require extra memory bandwidth as they process data which are already in the onchip memory. (In the dissertation, an accelerator with one FPGA board was considered, in which case the processor concatenation has no technical limitations.)

DOI:10.15774/PPKE.ITK.2014.001

DOI:10.15774/PPKE.ITK.2014.001

Chapter 4

Generating Arithmetic Units: Partitioning and Placement

The demand for a high-performance arithmetic unit in the PDE solver, presented in the previous chapter, motivated the improvement of the design methodology of pipelined AUs and the automation of the mapping of mathematical expressions into FPGAs.

In Section 4.1, a locally distributed control unit is presented which can operate at higher frequency than the global control unit. In Section 4.2, a graph partitioning problem is proposed to determine the locally controlled parts of the arithmetic unit. In Section 4.3, frequently used graph partitioners are reviewed, while the advantage of the proposed control is empirically validated via a new greedy algorithm in Section 4.4. In Sections 4.5, the partitioning objectives are improved and extended with placement objectives. A simulated annealing based algorithm is given to address all the objectives and to improve the operating frequency of the resulting circuit. A framework is described, which was implemented in C++ to automatize the mapping procedure and to test the partitioning algorithms. The section concludes with the evaluation of the implementations results of the two CFD problems.

4.1 Locally distributed control of arithmetic unit

4.1.1 The proposed control unit

In the arithmetic unit the mathematical expression which describes the flux crossing the interface between two adjacent cells is implemented. All input and output variables

DOI:10.15774/PPKE.ITK.2014.001

of the AU are stored in separate FIFO buffers. The AU is designed to operate independently from the rest of the processor and start the computation of a new interface in each clock cycle if all the inputs are available. To design an efficient *control unit* (CU) to the AU, the fanout of the control signals and the LUT depth of the control logic shall be minimized, otherwise the wire delays will hold down the operating frequency of the whole AU [26].

One straightforward way to control the AU is to implement a global CU, which checks the states of the input FIFOs and schedules the operation of the FPUs. In this case every FPU is connected to the CU with a global enable signal, which can start or stop the operation of the given FPU. Unfortunately, in this case the fanout of the control signal and the complexity of the control logic are too high to reach the desired operating frequency.

If the AU does not have feedbacks and accumulators, the enable signal can be neglected to decrease the complexity of the CU. Instead of halting the FPUs, they are let to operate in full time, and the valid results are filtered out at the outputs of the AU. Filtering can be achieved by implementing an extra shift register, in which the pipeline stages which hold valid data are marked. As the FPUs cannot be halted and the data goes through the AU once it has been read, the output FIFOs have to be checked before the AU reads the inputs whether they will be ready to store the results of the AU. This can be solved by adding extra virtual FIFOs (one per each output FIFO) whose lengths are set to the length of the corresponding output FIFO. The usage of extra shift register and the virtual FIFOs are demonstrated in Figure 4.1.

Before the operation starts, the virtual FIFOs are empty, indicating all the corresponding output FIFOs and the pipeline are empty. In every clock cycle if the inputs are ready to be read and the virtual FIFOs are ready to be written, new input is read from the input FIFOs and a bit is written to each virtual FIFO indicating the number of occupied elements in the pipeline, and the corresponding output FIFO has been increased by one. The bit remains in the virtual FIFO as long as the data is in the pipeline or the corresponding result is in the corresponding output FIFO. This mechanism guarantees that every data which has entered the pipeline can be safely written out to the output FIFOs. To be able to read input into the pipeline in every clock cycle the size of the output and virtual FIFOs have to be at least the length of the pipeline.

DOI:10.15774/PPKE.ITK.2014.001

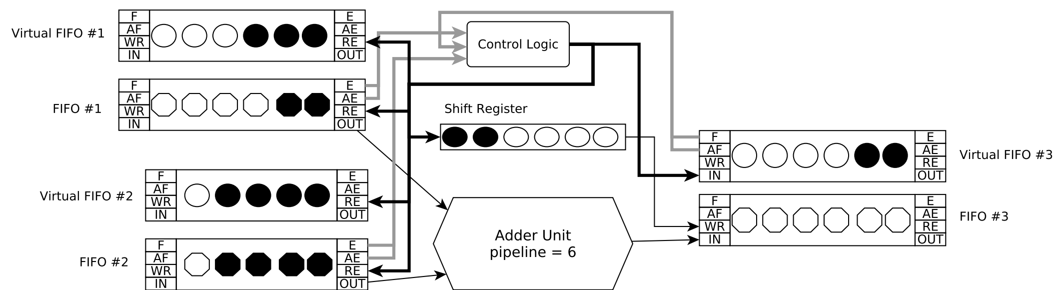

Figure 4.1: Usage of the shift register and the virtual FIFOs in case of a simple AU which contains only one adder FPU. Shift register is used to mark pipeline stages which hold valid data. In the example the first and second stages hold valid data. After 5 and 6 clock cycles the output of the shift register will write the results of the FPU to the output FIFO. Virtual FIFO #3 contains two elements indicating the two valid pipeline stages and will allow input data to enter the pipeline four more times if the output FIFOs is not read.

Without enable signal, the complexity of the CU can be significantly decreased and the fanout depends only on the number of I/O FIFOs of the AU. In case of simple mathematical expressions, the fanout is low, and the FIFOs can be placed close to one another and can be controlled at the desired frequency. However, in case of more complex expressions, the area requirement of the AU significantly increases and the fanout of control signals and the placement of the I/O FIFOs become critical. As the area requirement of the AU is affected by the floating-point precision we choose, the placement is even more challenging when 64 bit precision is applied (see results in Section 4.4.3). To reach the desired operating frequency, the FPUs shall be partitioned into separately controlled clusters which have smaller number of I/Os than the original AU. The partitioning problem can be described as the partitioning of the data-flow graph generated from the mathematical expression. If the arcs cut by the partitioning are replaced by FIFO buffers, the previously presented CU can be used for controlling each cluster. Unfortunately, the partitioning of the FPUs introduces other problems which have effects on both the area and the operating frequency of the circuit.

First of all, the added synchronizing FIFOs explicitly increase the area requirement of the circuit, which makes the efficient placement of the AU more challenging. To minimize the area requirements, the number of cut arcs shall be minimized.

To explain the second problem, a partitioned data-flow graph and the corresponding *cluster adjacency graph* are shown in Figure 4.2, where clusters are represented

DOI:10.15774/PPKE.ITK.2014.001

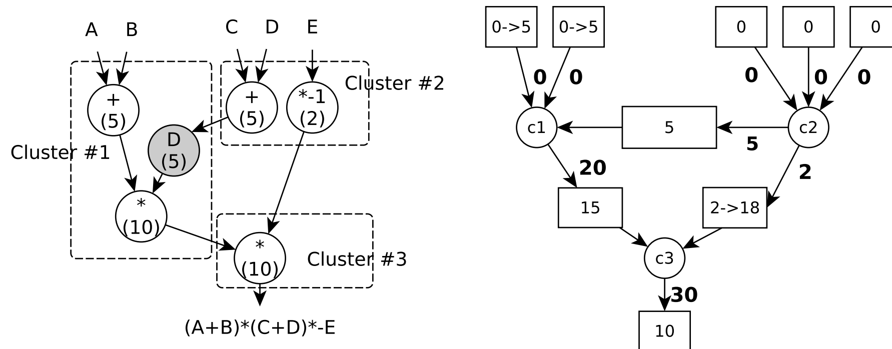

Figure 4.2: On the left a partitioned data-flow graph, while on the right the corresponding cluster adjacency graph is shown. In the data-flow graph pipeline length of the FPUs is displayed in brackets. In the cluster adjacency graph, a FIFO (indicated by rectangle) is added to every cut arc. In cluster 1 an extra delay shift register has to be added to keep the proper data timing inside the cluster. The length of the FIFO between cluster 2 and 3 has to be set to 18 instead of 2, otherwise cluster 2 cannot operate continuously.

by vertices and the connections between the clusters (cut arcs) are represented by arcs. The position (level) of an arc in the pipeline is defined as the first clock cycle when partial results computed from the first inputs reach the given arc. In the cluster adjacency graph, the levels of the arcs and the size of the added FIFOs are also displayed. If a cluster (e.g. cluster 3) has two inputs (two inward cut arcs) which have different levels, that is, the partial results reach the given cluster in two different routes, the data arriving at the shorter route has to be stored until other parts of the input arrive. In the presented example, if the length of the FIFO between cluster 2 and 3 was only 2, the operation of cluster 2 would be paused after every 2 successful reads for 18 clock cycles, which is the time needed for the data to reach cluster 3 via the other route. The size of a FIFO at a given input arc shall be set at least to the level of the highest level input arc minus the level of the given arc to guarantee continuous operation. The size of the introduced FIFOs and the overall pipeline length of the AU heavily depend on the partitioning, therefore, an ideal data-flow graph partitioner shall avoid clusters which have big differences in the levels of the incoming arcs. Unfortunately, common partitioning algorithms, which minimize the number of cut arcs, cannot target this objective. In the illustration, the minimum size of the FIFOs is given, however, in practice, these values are rounded up to the nearest number which is integer power of

DOI:10.15774/PPKE.ITK.2014.001

2 because they can be implemented more efficiently in FPGA.

The third problem is that the partitioning can create directed cycles in the cluster adjacency graph (mutually dependent clusters) even if the data-flow graph is acyclic. As a cluster reads new input only if all of its input FIFOs are ready to be read, mutually dependent clusters will never start reading, and cause a deadlock in the AU. Unfortunately, common partitioning algorithms do not have the mechanism to avoid mutually dependent clusters.

In the following sections, two partitioning algorithms are given to tackle these problems. The first one is a simple greedy algorithm (presented in Section 4.4), which addresses only the first problem, to validate the implementation benefit of the constrained partitioning. The second one is a more complex algorithm (presented in Section 4.5) which addresses all the problems and gives a robust solution from all aspects.

4.1.2 Trade-off between speed and number of I/Os

The maximum operating frequency of the proposed CU is affected by the fanout and the LUT depth of the controlling signals. Consequently, it is determined by the number of I/Os of the controlled cluster. There is a trade-off between the speed of the CU and the number of the I/Os. In practice, the operating frequency of the slowest FPU determines a minimum operating frequency demand for the CUs. It is not worth to design a significantly faster CU, however, each CU should be able to operate at least at this frequency. From an engineering point of view, the question is the following: what is the maximum number of the I/Os that guarantees the desired frequency?

To be able to answer the above-mentioned question, I implemented the proposed control unit with different number of FIFOs attached to it and with different seed parameters of the place-and-route process. The measurements were done for a Virtex-6 SXT FPGA, and for each number of I/Os the highest frequency was selected. According to the results, the control unit can handle 10 input/output FIFOs without approaching the 450MHz operating frequency of the multiplier unit (see Figure 4.3). In the measurements, empty clusters were considered without real FPUs, however, in practice, the FPUs affect the placement of the FIFOs resulting in smaller operating frequencies. In my first experiments, for safety reasons, the I/O limit of the clusters was

DOI:10.15774/PPKE.ITK.2014.001

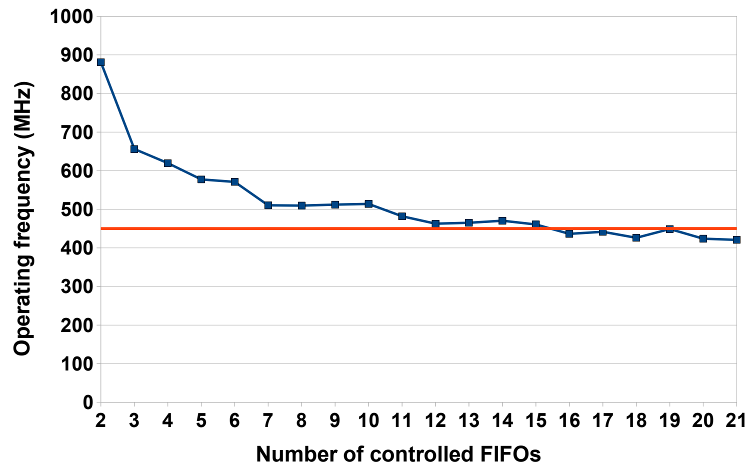

Figure 4.3: Operating frequency of the proposed control unit. Red line indicates the operating frequency of the multiplier unit.

chosen to 10, however, in case of the partitioning algorithm described in Section 4.5.2, the effect of I/O limit was investigated for other values as well.

4.2 Partitioning problem

In the section, a high-level partitioning problem is described where the data-flow graph representation of a mathematical expression is partitioned to determine the locally controlled parts of the resulting arithmetic unit.

4.2.1 Problem formulation

After converting the mathematical expression into a data-flow graph (directed acyclic hypergraph), the number of cut arcs can be minimized by graph partitioning techniques, and the size and I/O connections of the clusters can also be balanced or constrained. The number of cut arcs shall be minimized to reduce the area requirements of the circuit, while constraining the number of I/O connections of the clusters provides high-speed control units. Mathematical foundation of the problem is established in the

DOI:10.15774/PPKE.ITK.2014.001

following paragraphs. Definitions are stated in accordance with [27].

Definition 1 *A directed hypergraph denoted by $G(V, E)$ is a pair $< V, E >$, where V is a non empty set of nodes (vertices) and E is a set of hyperarcs (hyperedges); a hyperarc e is an ordered pair $< S, T >$, with $S \subset V$, $S \neq \emptyset$, and $T \subset (V \setminus S)$. Elements of S and T are called the sources and targets of the arc, respectively. Sources and targets of a hyperarc e is denoted by $S(e)$ and $T(e)$, respectively.*

In a hypergraph $G(V, E)$, a path p_{st} between nodes s and t is an alternating sequence of distinct nodes and hyperarcs $s = v_0, e_1, v_1, e_2 e_k v_k = t$ such that $v_{i-1} \in S(e_i)$ and $v_i \in T(e_i)$ for all $i = 1...k$. A path p_{st} is called a cycle if $s = t$. A directed hypergraph is called acyclic if there is no cycle in the hypergraph.

In our case, the data-flow graph of the high-level circuit can be mapped to a special acyclic hypergraph, in which each hyperarc e has only one source: $|S(e)| = 1$. The special property comes from the fact that every signal is driven by one source in the design. The acyclic property comes from the assumption that we deal with a simple evaluation of a mathematical expression which can be implemented without accumulators or recursion.

Definition 2 *Given a hypergraph $G(V, E)$, a P decomposition of V into disjoint subsets $V_1, V_2, ..., V_n$ such that $\bigcup_i V_i = V$ is called a partitioning of G. The terms subdomain, cluster, or partition class are used to refer to each one of these V_i sets.*

The proposed optimization can be described as a hypergraph partitioning with special cost functions assigned to cut hyperarcs or partition classes. In the presented model, an *area cost* is defined for each cut arc describing the number of required FIFOs:

$$f_{Area}(e, P, G) := |\{j : (S(e) \cup T(e)) \cap V_j \neq \emptyset\}| - 1 \tag{4.1}$$

Each cut arcs, which have targets in classes different from the source class, shall be replaced by one FIFO for each target class. The area cost of a partition P can be computed as the sum of the cost of the arcs.

$$F_{Area}(P, G) := \sum_{e \in E} f_{Area}(e, P, G) \tag{4.2}$$

DOI:10.15774/PPKE.ITK.2014.001

To formulate the IO constraint, a *control cost* is defined for each arc and partition class:

$$f_{Control}(e, V_i, P, G) := \begin{cases} 0 & \text{if } (S(e) \cup T(e)) \cap V_i = \emptyset \\ 1 & \text{if } S(e) \not\subset V_i \text{ and } T(e) \cap V_i \neq \emptyset \\ f_{Area}(e, P, G) & \text{if } S(e) \subset V_i \text{ and } T(e) \not\subset V_i \end{cases} \tag{4.3}$$

If an arc is completely outside a class, the control cost for the class is zero. If an arc has only targets in a class, the control cost for the class is one. Finally, if an arc has the source in a class, the control cost equals the area cost, because all added FIFOs will be controlled by the class. The control cost of a partition class can be computed as the sum of the cost of the arcs:

$$F_{Control}(V_i, P, G) := \sum_{e \in E} f_{Control}(e, V_i, P, G) \tag{4.4}$$

The relation $F_{Area}(P, G) * 2 = \sum_{V_i \in P} F_{Control}(V_i, P, G)$ can be concluded by observing the fact that every FIFO is connected to two control units. Hence, every FIFO is computed twice in control cost computation. As a consequence of the relation, either cost function can be minimized for our purposes, however, the constraining cannot be skipped, otherwise the control costs of some classes may exceed the user defined limit.

The mathematical formulation of the proposed partitioning can be given as the following constrained optimization:

Problem 1

$$\begin{aligned} \underset{P}{\textbf{\textit{minimize}}} \quad & F_{Area}(P, G) \\ \textbf{\textit{subject to}} \quad & F_{Control}(V_i, P, G) \leq c, \; V_i \in P, \\ & V_1 := \textit{global inputs}, \\ & V_2 := \textit{global outputs} \end{aligned}$$

where c is an upper limit for the number of FIFOs controlled by one control unit (proposed in Section 4.1.2). Note that two partition classes containing the global input and output nodes of the data-flow graph are fixed to avoid a trivial solution in which all vertices belong to the same partition class. Also note that the data-flow graphs where the number of global I/Os is less than the previously described upper limit can be implemented with one control unit efficiently, and there is no need for optimization.

DOI:10.15774/PPKE.ITK.2014.001

Digital circuits are usually described by *netlists*, which precisely enumerate the pin connections of each net of the circuit. According to the objective functions used in the partitioning, the netlist representation can be simplified to a more simple representation (e.g. hypergraphs, directed graphs). Netlists can be represented by hypergraphs if vertices and hyperedges are assigned to circuit modules and nets, respectively. In this case pins belonging to the same module are not distinguished and hyperedges naturally represents nets, as they can connect more than two modules. Hypergraph representation can be further simplified to graphs, however, in this case hyperedges have to be modeled by extra edges and vertices [28]. In the clique net model, each hyperedge is replaced by edges connecting each pair of vertices incident to the given hyperedge. In the directed graph model, each source vertex of a net/hyperedge is connected to each target vertex of the hyperedge via a new edge. In the bipartite graph model, each hyperedge is replaced with a new vertex, and old and new vertices are connected if the corresponding vertex was incident to the corresponding hyperedge. Unfortunately, in case of most partitioning objectives, hyperedges cannot be equivalently replaced [28]. Similarly, in case of the F_{Area} objective, I have not found such a replacement for hyperedges which does not alter the F_{Area} objective function. From the aspect of the proposed optimization problem, both the fan-out and the directions of nets are important, which explains the application of the more complex hypergraph model.

Finding an optimal solution for Problem 1 is NP-complete because if we could solve this problem in polynomial time, it would yield a polynomial algorithm for the ”graph partitioning problem” (Problem ND15 in [29]), which is known to be NP-complete. For a proof, note that any normal graph is also a hypergraph and the effects of I/O constraints can be eliminated by selecting a very large upper limit.

4.3 Partitioning algorithms used in circuit design

As the partitioning problem is NP-complete, several heuristics have been proposed over the last few decades. The first few heuristics (e.g. [30]) were motivated by circuit design, however, later, the problem arose in several other research areas, such as unstructured grid computations or sparse matrix operations. As the parallelization of the most time-consuming applications produced larger and larger graphs, the partitioning

DOI:10.15774/PPKE.ITK.2014.001

programs started to focus on large-scale partitioning, which led to the introduction of the multilevel paradigm.

Hereby, I review the most common heuristics, partitioning techniques, and software packages in accordance with [31–33]. In Section 4.4 and 4.5, these works will be related to the algorithms I propose in the dissertation.

The partitioning algorithms can be grouped into *single-* and *multilevel* algorithms. In the beginning, single-level algorithms had been proposed, such as move-based and spectral partitioning algorithms. Later, when multilevel paradigm arose, several multilevel algorithms were presented, which usually utilized some of the single-level algorithms in a multilevel framework to reach better quality and higher speed for large graphs. Nowadays, the multilevel paradigm is the de facto standard [32] even for smaller problems, however, one of the main drawbacks of the technique is that it is hard to integrate complex objective functions and constraints into the model. The objective functions and constraints used in the dissertation also belong to this category, and their integration into the multilevel framework is not trivial (see Section 4.5.3 for more details). Furthermore, as the relatively small size of the graphs generated from mathematical expressions does not predict significant benefit, the multilevel approach has not been addressed yet in my work.

4.3.1 Move-based heuristics

I shortly review two of the oldest heuristics, which are still frequently used in the state-of-the-art partitioners. These algorithms originally focused on the rather simple cutnet metric (see Equation 4.5), however, later several extensions [32] have been proposed to handle very complex objectives, constraints and hypergraphs.

4.3.1.1 The Kernighan-Lin algorithm

The original Kernighan-Lin (KL) algorithm [30] is an iterative swap based heuristic to 2-way partition a graph with low number of cut nets. The cost of a cut net is calculated according to Equation 4.5.

$$f_{cutnet}(e, P, G) := \begin{cases} 0 & \text{if } \exists i : e \subseteq V_i \\ 1 & \text{otherwise} \end{cases} \tag{4.5}$$

DOI:10.15774/PPKE.ITK.2014.001

The algorithm starts with an initial partitioning and is executed through passes, which consists of iterations. In each iteration, all the vertices which were not swapped in the current pass are considered for swapping, and the pair with maximal gain is swapped. Iterations of a pass end when all the vertices are swapped. At the end of the pass, the sequence of swaps are revisited and some of the last swaps are reverted to reach the state where the objective function was minimal. As swaps with negative gain are also allowed in the iterations, the algorithm can escape from some of the local minima.

The runtime of a pass of the algorithm has a complexity of $O(n^3)$, where n is the number of the vertices of the graph. The runtime is typically dominated by the $O(n^2)$ gain updates of the vertices, and the $O(n^3)$ comparisons of the vertex gains used to determine the best pair for the swapping.

4.3.1.2 The Fiduccia-Mattheyses algorithm

The Fiduccia-Mattheyses (FM) algorithm [34] can be regarded as an improved KL algorithm and has several features compared the original algorithm. The most important difference is that instead of swapping of vertices one vertex is moved across the cut in each iteration. The modification enables the algorithm to handle unbalanced partitions, in which a desired ratio of the size of the clusters can be user-defined. Furthermore, a straightforward k-way extension of the algorithm is possible [35] to partition the graph into k subdomains directly. In recent modifications, both the data structure and the gain computation are changed, however, the framework of the algorithms are the same.

In case of hypergraphs, the definition of *critical nets* can be introduced, which simplifies the gain calculation for the vertices. A net is critical if the movement of one of its vertices can change the cutnet metric, that is, the net has a vertex which is associated to a cluster different from the ones containing the rest of the vertices. By defining the $FS()$ and $TE()$ functions, the gain difference for the movement of vertex v can be computed as

$$\triangle\ gain(v) = FS(v) - TE(v) \tag{4.6}$$

where $FS(v)$ indicates the number of nets which include only v from the vertices of its cluster, and $TE(v)$ denotes the uncut nets connected to v. The positive gain indicates the decrease of the overall cutnet metric, therefore, the vertex with the highest

DOI:10.15774/PPKE.ITK.2014.001

gain is the best candidate for movement, although the balance constraints shall also be checked.

The runtime of a pass of the algorithm has a complexity of $O(P)$, where P is the total number of the *pins*, also known as the degrees of vertices. The linear complexity is due to the so called *bucket list* representation and a very efficient neighborhood update scheme. The special representation is the result of the fact that the gain of any move is bounded between plus and minus the maximal vertex degree. As a consequence, unvisited vertices can be stored in buckets and the gain updates are carried out by moving vertices between the buckets in constant time. Finally, in case of the original 2-way partitioning, the number of vertices to be updated can be efficiently limited. In the naive approach every neighbor of the moving vertex should be updated resulting in $O(P^2)$ complexity, however, if we update only those vertices where critical nets are affected this reduces to $O(P)$.

4.3.2 Spectral partitioning

Spectral partitioning methods are also termed as geometric representation methods, because at first the graphs are converted into a geometric representation, then well-known geometric heuristics are used to carry out the partitioning. Essentially, the success of a spectral partitioning algorithm depends on the quality and speed of the conversion and the geometric heuristic.

Hereby, I review the basic concepts of spectral partitioning by describing the original spectral bipartitioning technique and one of its complex descendants, which can handle multi-way partitioning of hypergraphs as well. Although, in the beginning, the spectral partitioning attracted a lot of research efforts, its performance was not proved to be superior to the alternative iterative techniques in case of large graphs [36]. Furthermore, the spectral technique is not flexible enough for real word problems as it is hard to integrate more complex objectives and constraints [37] into the geometric representation. As a consequence, the multilevel move-based techniques are considered better alternatives in modern circuit design technology.

DOI:10.15774/PPKE.ITK.2014.001

4.3.2.1 Spectral bipartitioning

The first spectral bipartitioning heuristics were motivated by the work of Fiedler [38] and Hall [39] to partition ordinary graphs with the simple cutnet metric. The theory is based on the special properties of the *Laplacian matrix* of a graph.

The Laplacian matrix of a $G(V, E)$ graph is a matrix with $|V|x|V|$ size and defined as

$$L = D - A \tag{4.7}$$

where D is the *degree matrix* of the graph containing the degrees of the vertices in its diagonal, and A is the *adjacency matrix* of the graph. The *degree* of a vertex is defined as the number of the incident edges, while the value of an a_{ij} element of the adjacency matrix is the number of the edges connecting v_i and v_j vertices. The specialty of L is that the sum of its rows is always zero. The l_{ii} diagonal element corresponding to vertex v_i contains the maximum cutnet which can be related to v_i, while the other l_{ij} elements in the row indicates the benefit of putting v_i and v_j to the same partition class. The presented definitions can be naturally extended for weighted graphs as well.

The Laplacian quadratic form can be defined as

$$x^T L x \tag{4.8}$$

where is x is a vector of $R^{|V|}$. In the original bipartitioning scenario, only one x vector is considered, and it is imagined as a *partitioning vector* containing ones and zeros to describe one of the two clusters of the partitioning. In this case the quadratic form equals to the number of edges cut by the cluster (see Equation 4.9), as in the quadratic form, the number of uncut edges inside the cluster are subtracted from the maximum possible cutnet of the vertices residing in the cluster.

$$x^T L x = \sum_{e \in E} (x(u) - x(v))^2 \tag{4.9}$$

Observing Equation 4.9, it is obvious that it cannot be negative. Furthermore, for all *constant vectors* (which contain the same constant for each of their entries) the quadratic form equals to zero. In other words, the smallest eigenvalue of L is always zero and constant vectors constitute the corresponding eigenspace. A constant vector can represent a trivial solution, when we put all the vertices into the same cluster.

DOI:10.15774/PPKE.ITK.2014.001

For bipartitioning, the second smallest eigenvalue λ_2 (also called *algebraic connectivity*) and the corresponding eigenvector (also called *Fiedler vector*) is more interesting. It is easy to see that the $\lambda_2 > 0$ if and only if the graph is connected and the $x \in R^{|V|}$ Fiedler vector gives a 1D representation of the vertices in which the squared distance of the endpoints of the edges are minimized (see Equation 4.9). Consequently, the representation can be used to split the graph e.g. at the largest gap between two vertices, however, there is no guarantee that the continuous solution closely approximates the solution of the $x \in N^{|V|}$ discrete case.

4.3.2.2 Spectral partitioning with multiple eigenvectors

In the last decades of the 20th century, several improvements and extensions of the original spectral bipartitioning were proposed [28] to adapt the algorithm for multi-way hypergraph partitioning. One of the most successful modification was the work of Alpert [40] that proposed to use as many eigenvectors of the Laplacian as possible. Hypergraphs were converted to ordinary graphs using the *clique net model*, in which each hyperedge is replaced by weighted edges connecting each pair of the vertices of the hyperedge, although it was proved earlier that there is no universal weighting scheme for the clique net model that can perfectly realize the cutnet metric [28].

The k-way partitions were represented by the *indicator matrix*, which is a $|V|xk$ matrix containing the partitioning vectors mentioned before. Using the indicator matrix in the quadratic form, the cutnet metric appears in the trace of the result, and the following equation can be reached via the $L = U\Delta U^T$ eigenvalue decomposition of L.

$$F_{cutnet}(G) = tr\{X^T L X\} = tr\{\Gamma^T \Delta \Gamma\} = \sum_{i=1}^{k} \sum_{j=1}^{n} \alpha_{ij}^2 \lambda_i \tag{4.10}$$

where $\Gamma = (\alpha_{ij})$ is defined as $\Gamma := U^T X$.

One of the main discoveries used in the method was that the partitioning problem with the F_{cutnet} objective can be mapped to a *max-sum vector partitioning problem*. A k-way partitioning $S^k = \{S_1, S_2, ..., S_k\}$ of n d-dimensional vectors $Y = \{y_1, y_2, ..., y_n\}$ is defined as k disjoint subsets of Y, such that $S_1 \cup S_2 \cup ... \cup S_k = Y$, while the max-sum objective is defined as

$$g(S^k) = \sum_{S_i \in S^k} \|Y_i\|^2 \qquad where \quad Y_i = \sum_{y_h \in S_i} y_h \tag{4.11}$$

DOI:10.15774/PPKE.ITK.2014.001

If the y_i vectors to be partitioned are the rows of the $\Delta^{\frac{1}{2}}U$ matrix, the two objectives (Equation 4.10 and 4.11) are the same:

$$F_{cutnet}(G) = g(S^k) \tag{4.12}$$

The proof can be constructed by observing the $\|Y_i\|^2 = \sum_{j=1}^{|V|}(\sqrt{\lambda_j}U_j^T X_i)^2$ equality, where U_j is the j_{th} column of matrix U, and X_i is the i_{th} column of the indicator matrix X.

In [40] the standard k-means clustering method [41] was utilized to address the max-sum vector partitioning, however, as it is a maximization procedure, the task was reformulated to maximize the $|V|\lambda_{max} - F_{cutnet}$ objective.

4.3.3 Simulated annealing

The *simulated annealing* (SA) technique was first described by Kirkpatrick et al. [42] in 1983. The technique quickly became a popular alternative to greedy algorithms as it is capable of making uphill movements to avoid local minima. The technique starts from an initial solution, and then, in each iteration, a random neighbor of the current solution is picked for the evaluation of the cost function. Through the iterations, a T temperature parameter is maintained and decreased continuously, which determines the so called *Boltzmann acceptance rule* for uphill movements; SA accepts every uphill movement of the cost with probability $e^{-\Delta/T}$, where Δ indicates the increase of the cost.

One of the first works considering simulated annealing for partitioning was [43], which found the technique competitive to move-based heuristics in random graphs. On the one hand, the technique has several advantages. First, it is very flexibility to integrate complex objectives and constraints. Second, it can be shown that SA converges to the global optimum solution given an infinite number of moves if the temperature is decreased sufficiently slowly [44]. On the other hand, the speed of its convergence was turned out to be slower than rival heuristics in case of large graphs required in real world applications [28], and consequently, the technique is not part of the techniques currently used in circuit partitioning.

From the aspect of my research aims, the flexibility of the technique had a greater significance than the speed of the convergence, because the relatively small size of

DOI:10.15774/PPKE.ITK.2014.001

the investigated graphs tolerates the slow convergence. The main challenge of partitioning problem proposed in the dissertation is to balance between complex objective functions and constraints, therefore, the simulated annealing is an appealing candidate to demonstrate an algorithm capable of handling all the required objectives and constraints. The algorithm proposed in Section 4.5 successfully utilized the simulated annealing technique to address all the objectives and constraints required to produce high-performance locally controlled arithmetic on FPGA.

4.3.4 Software packages incorporating the multilevel paradigm

Inspired by the success of multigrid methods used in other research areas, the multilevel paradigm for graph partitioning was introduced by Barnard et al. [45]. The first multilevel implementation was a recursive spectral bipartitioning algorithm designed for partitioning large unstructured meshes for distributed-memory architectures. The first attempt was later followed by two famous software packages, Chaco [46, 47] and Metis [48], which made the multilevel approach a quasi standard in graph partitioning. Although the two solutions differed in details, they shared the multilevel approach and produced competitive results. Over the years, both software packages was followed by new editions, and also the group of multilevel partitioners was extended with new software packages (e.g. Scotch [49], JOSTLE [50], Parkway [51]). The research group who developed the Chaco program created a more powerful software package, called Zoltan [52], to support high-performance parallel partitioning, parallel graph coloring and dynamic load balancing. The Metis software package was followed by hMetis [53], which was designed for partitioning hypergraphs arising in VLSI circuits, and ParMetis [54], which was created for MPI-based parallel execution of partitioning of large graphs.

The focus of the partitioning programs moved to parallel execution to keep up with the increasing size of the graphs challenging the modern computing architectures. As the size of the graphs investigated in the dissertation is relatively small, the lack of parallel execution can be tolerated. Hereby, I review the algorithms of the two most famous software packages (Chaco and hMetis), which still provide the basis of more complex parallel partitioning packages. To demonstrate the limitations of naive partitioning for the problem proposed in the dissertation, hMetis was selected as a represen-

DOI:10.15774/PPKE.ITK.2014.001

tative multilevel algorithm and compared to the algorithm I proposed in Section 4.4.1. From the aspect of combined partitioning and placement, the terminal propagation feature of Chaco is also noted, and the technique is related to the algorithm proposed in Section 4.5.

4.3.4.1 Chaco

The Chaco [47] software package is based on the spectral bipartitioning technique I reviewed in Section 4.3.2.1. The key contribution of the work is to apply the multilevel paradigm to spectral bipartitioning. As a typical multilevel algorithm, it has three phases: *coarsening*, spectral partitioning, and *uncoarsening* with local refinement. The key success of the algorithm is that the cost of both coarsening and uncoarsening phases is very low and proportional to the number of edges. During coarsening, instead of eigenvectors, only the partition is transferred to the next level. The relatively expensive spectral partitioning is carried out only on the coarsest graph, which has a very limited size. To preserve constraints related to cluster size or weighted edges, the weight of vertices and edges can also be adjusted at coarsening.

As the algorithm is designed for ordinary graphs, at coarsening, edges containing only two vertices has to be contracted. When an edge is selected for contraction, the incident two vertices are joined, and the weight of the new vertex equals to the sum of the weights of the original vertices. Additionally, the edges, which connected the original vertices to a common vertex, are joined, and the weight of the new edge equals to the sum of the weights of the original edges.

In the beginning of each coarsening step, a *maximal matching* is generated to determine the edges to be contracted. A maximal matching of a $G(V, E)$ matrix is the maximal subset of E edges, in which no two edges share a common vertex. In the program, the maximal matching is generated via visiting the edges in a random order, which requires a time proportional to the number of edges.

At the uncoarsening phase, both vertices and clusters are projected back to the previous level. As the back-projected partition is not necessarily at a local optimum, a local refinement algorithm (Fiduccia-Mattheyses described in Section 4.3.1.2) is applied for fine-tuning.

DOI:10.15774/PPKE.ITK.2014.001

An interesting feature of the program is the *terminal propagation*. The terminal propagation was originally proposed by [55] and later integrated into the spectral partitioning technique [56] as well. It was inspired by the problem of partitioning meshes for parallel architectures with fixed topology (e.g. hypercube with different dimensions). A special partitioning was required which discriminates against cut edges (communications) that connect nonadjacent clusters (computing nodes). The idea was to add virtual vertices to represent the already formed clusters of the recursive bipartitioning. For each unpartitioned vertex, a virtual edge was added to the graph connecting the vertex with one of the virtual vertices. Via the virtual edges, the preference to put a vertex close to an already formed cluster could be incorporated into the model for the rest of the bipartitionings.

Chaco supports partitioning for two types of architectures: hypercube and grid. In both cases, the possible dimensions are 1, 2, and 3. In case of grids, the exact size of the grid has to be specified, that is, the number of clusters has to be known before the partitioning. The terminal propagation is applied at both the recursive spectral bipartitioning of the coarsest graph and the move-based local refinements.

The constraint of a fixed parallel architecture is partly similar to the problem presented in the dissertation. In both cases, the penalty of cut edges depends on the topology of the clusters. The difference is that in my case the topology is not fixed. For a given graph (mathematical expression), it is not known a priori how many clusters it should contain or which cluster topology is the best for the maximum performance. Leaving these parameters free in the optimization procedure, one can answer these questions and reach better quality. In this sense, the algorithm proposed in Section 4.5.2 can be regarded as smarter technique addressing the problem without compromise. Although, accepting these compromises, an algorithm could be designed using the terminal propagation technique. A possible application of the technique is discussed and related to my solution in Section 4.5.2.6.

4.3.4.2 hMetis

The hMetis [53] software package is one of the state-of-the-art partitioning programs used for VLSI circuit partitioning. It is based on a multilevel hypergraph partitioning scheme and it is capable of minimizing several objective functions. The multilevel idea is to create a sequence of successive approximations of the original hypergraph and to

DOI:10.15774/PPKE.ITK.2014.001

partition a small-sized approximating hypergraph instead of the original hypergraph. The key contribution of the program is the approximation (called *coarsening*), when a smaller hypergraph is created from a hypergraph, and the inverse of this process (called *uncoarsening*), when a partition of an approximating hypergraph is projected back to the original hypergraph and the partition in the original hypergraph is *refined*. The partitioning of the approximating hypergraph can be done via a standard partitioning algorithm like Kernighan-Lin (described in Section 4.3.1.1) or Fiduccia-Mattheyses (described in Section 4.3.1.2).

For coarsening, uncoarsening, and refinement, several strategies are available in the hMetis program. In each case the objective is to find and contract such vertices at coarsening which would belong to the same partition class anyway. Common coarsening heuristics are *Edge Coarsening* (EC), *Hyperedge Coarsening* (HC) and *Modified Hyperedge Coarsening* (MHC). In EC special weights are added to hyperegdes ranking smaller sized edges higher, then pairs of vertices which are incident to hyperedges with the largest weights are contracted. In HC hyperedges are sorted in a decreasing weight order, and hyperedges of the same weight are sorted in a increasing size order, then vertices of full hyperedges are contracted. MHC is an enhanced version of HC: after contraction, the list of remaining hyperedges is traversed again and uncontracted vertices of the same hyperedge are also contracted. Uncoarsening simply projects the partition back to the original hypergraph, while refinement fine-tunes the quality of the back-projected partition via another partitioning (e.g. FM again).

4.4 Empirically validating the advantage of locally controlled arithmetic units

In the section, a greedy algorithm is given and the state-of-the-art hMetis algorithm is also utilized to address the problem described in Section 4.2. Both algorithms are tested via the presented structured CFD simulation. Vertices, representing the operators of the expression, are implemented via Xilinx IP cores, and clusters are controlled via the control unit described in Section 4.1. The resulting VHDL representation of the circuit is implemented with the standard Xilinx synthesis and place-and-route tools. Partitioning in both cases improves the operating frequency, however, in case of the

DOI:10.15774/PPKE.ITK.2014.001

greedy algorithm, a straightforward manual tuning of the placement constraints, which is not available in case of hMetis, produces a higher frequency.

4.4.1 The proposed greedy algorithm

The number of cut arcs can be minimized and the I/O connections of clusters can be evenly distributed by traditional graph partitioning techniques, however, the resulting operating frequency is far from the theoretical limits suggested by FPGA data sheets. According to my measurements, during the synthesis of a locally controlled AU, which was generated from a well-partitioned (see Problem 1) data-flow graph, high operating frequency is estimated, however, at place-and-route phase the circuit cannot be placed efficiently resulting in a significantly lower operating frequency.

The Xilinx place-and-route process gives the designer the ability to constrain the physical position of parts of the circuit. In my experience, constraining the placement of the partition classes and the synchronizing FIFOs indeed improves the operating frequency. Unfortunately, in case of naive partitioning techniques, the manual placement of the resulting classes and FIFOs is very challenging, and the operating frequency is sensitive to the local connectivity. In Figure 4.4 manual placement constraints of a circuit, which was partitioned with a naive partitioning technique [3], are demonstrated. Despite the fact that the partitioning limited the number of I/Os of each class, the poor placement resulted in low frequency. In the figure, the critical net (also indicated by red in Figure 4.6) explicitly limiting the operating frequency is colored by red. The question is how to place all connected components close to one another to avoid long interconnections. In the presented example the critical net is related to an input variable, which is used in three different operations in the mathematical expression. The three operations are associated to three different partition classes, therefore three extra synchronization FIFOs are required and should be placed close to one another, as they use the same input. (The phenomenon also exists at lower layers, see green vertices in Figure 4.6.) A smart partitioning algorithm shall avoid multiway cutting of hyperarcs and make the efficient placement possible.

The idea of the new algorithm, inspired by the previous problem, is to draw the graph into the plane before the partitioning starts. If a representation of the graph which minimizes the distance between the connected arcs is given, a simple greedy

DOI:10.15774/PPKE.ITK.2014.001

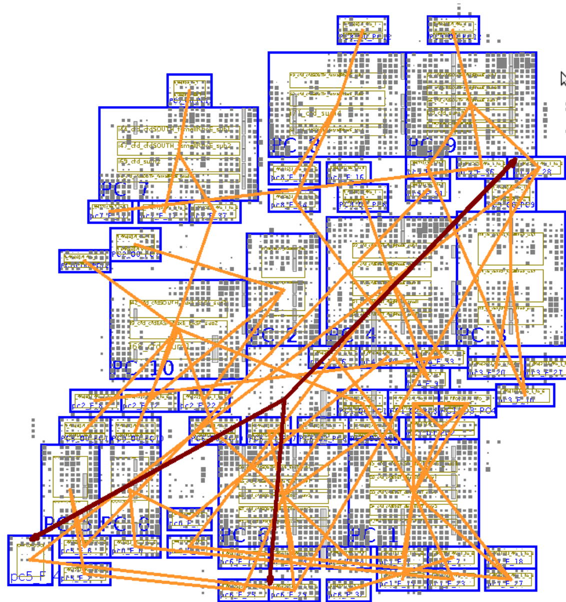

Figure 4.4: Placement constraints (blue) and the placed instances (grey) of a circuit generated from a partition which was created by a naive partitioning strategy. Connectivity of partition classes is indicated by orange while a timing critical net is indicated by red (also shown in Figure 4.6). The figure was generated with the standard Xilinx place-and-route editor. It demonstrates the challenges a developer has to solve during manual placement of circuit elements. In a high-performance implementation, connected elements shall be placed close to one another.

DOI:10.15774/PPKE.ITK.2014.001

algorithm can provide a partitioning without long interconnections. Furthermore, the placement becomes straightforward and placement constraints can be adjusted manually. The proposed strategy significantly differs from low-level partitioning and placement techniques as we partition and place the circuit at the level of IP components.

The proposed algorithm is a two-step procedure preceded by a simple preprocessing. During preprocessing, vertical coordinates of the vertices are fixed via a simple method called *layering*. Next, in the first step, horizontal coordinates are calculated to minimize the distance between the connected components. Finally, in the second step, a greedy method is used to partition the graph based on the spatial information of the vertices. The steps of the algorithm (also summarized in Algorithm 1) are described in the following paragraphs.

Algorithm 1 Outline of the proposed greedy algorithm.

1: Add delay vertices to make the graph bipartite, and associate every vertex with a level according to its distance from the global inputs (layering).
2: Place vertices randomly into the corresponding layer.
3: Create an initial horizontal placement of vertices by the barycentre heuristic.
4: Find the horizontal position of the vertices and a local minimum of the objective function via a simple swap-based iterative algorithm.
5: Partition the graph according to the spatial positions of the vertices using a greedy algorithm.

4.4.1.1 Preprocessing and layering

The mathematical expression to be implemented is described in a text file, where inputs, outputs and internal variables are also defined. The input file is parsed and a data-flow graph representation of the mathematical expression is created. Every mathematical operator is represented by a vertex and has an associated delay which will be the pipeline latency of the corresponding IP core in the implemented circuit.

In the next step a *layering* [31] is performed, in which the data-flow graph is converted to a special bipartite graph. In this bipartite graph, every vertex is associated to a layer and each arc directs immediately to the next layer.

DOI:10.15774/PPKE.ITK.2014.001

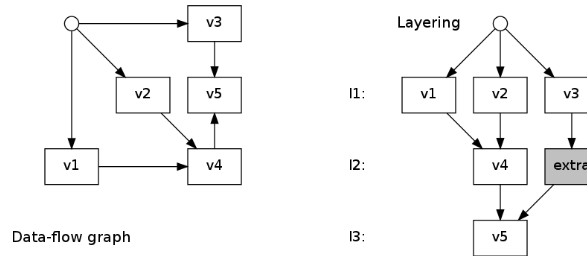

Figure 4.5: A simple data-flow garph and its layered version

Definition 3 $L = \{l_1, l_2, ...\}$ *layering is a partition of the* V *vertices of the* $G(V, E)$ *directed graph such that*

$$\forall e \in E : \qquad S(e) \subset l_i \textit{ and } T(e) \subset l_{i+1}.$$

A layering can be generated via a breadth-first-search in linear time [31] by splitting up the arcs which span more than one layer with extra delay vertices. An example layering is shown in Figure 4.5. The layering of the graph is an artificial restriction on the placement and the partitioning of the graph. Vertices can only be moved horizontally during the placement, and the representation of the clusters also depends on the structure of the layers. Evidently, this restriction can leave out the optimal solution from the search space, however, it significantly decreases the representation costs. Unfortunately, the complexity of the original problem requires some expandable restrictions, otherwise the problem cannot be handled.

Fortunately, layering has several other benefits beside the simplification. First of all, if the vertices had the same pipeline lengths, the horizontal cutting would guarantee that the incoming arcs of a cluster have the same pipeline level. Despite the pipeline lengths are different in practice, horizontal cutting combined with the cut arcs minimization produces acceptable results and does not increase the overall pipeline length drastically (see Table 4.3). The second benefit of the layering is that a simple and sufficient criterion can be formulated to check the existence of deadlocks during the simulated annealing (see Lemma 2).

Finally, in physical implementation, extra delay vertices are implemented as shift registers (extra vertices inside one cluster are joined), which hold the data for the proper number of clock cycles. From the aspect of performance, it is advantageous because smaller interconnections help to keep the timing requirements.

DOI:10.15774/PPKE.ITK.2014.001

4.4.1.2 Swap-based horizontal placement

Vertices get horizontal coordinates randomly, then the number of edge crossings is minimized to create an initial solution. Minimal edge crossing objective does not guarantee good placement but it was found to be a good initial solution for my vertex swapping iterative algorithm.

Barycentre heuristic [57] is a fast and simple algorithm to minimize edge crossings in layered directed graphs, however, various other min-crossing algorithms can be applied for this purpose [58]. The minimization of the edge crossing is NP-complete, even if there are only two layers [59]. The method operates layer-by-layer: in every iteration one layer of the graph is fixed and the vertices of the next layer is arranged. The horizontal coordinate (x_A) of each vertex (A) is chosen to the barycentre of its *neighborhood* from the fixed layer:

$$x_A := \frac{1}{|N_A|} \sum_{v \in N_A} x_v$$

where N_A denotes the set of vertices connected to vertex A from the fixed layer, and x_v denotes the horizontal coordinate of a vertex v.

For horizontal placement, an adapted KL algorithm is used to minimize the distance between the connected vertices. The objective function is defined as the sum of the distance between the connected vertices. The distance between two vertices is determined according to their horizontal coordinates:

$$distance(A, B) := \begin{cases} (x_A - x_B)^2 & \text{if A and B are connected} \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

where x_A and x_B are the horizontal coordinates of vertex A and B, respectively. Both horizontal and vertical coordinates are integer numbers. The physical size of the floating-point units are not considered in this representation and set to one. Vertical coordinates can be neglected as the graph is a layered graph.

In the adapted version of the KL algorithm, the objective function has been replaced by Equation 4.13. During iterations, instead of swapping vertices between partition classes, the position of neighboring vertices are swapped. Similarly to the original algorithm, it can escape some of the local minima, however, not all of them. As it is very sensitive to the initial placement of the vertices, the previously described Barycentre heuristic is used to initialize the placement of the vertices.

DOI:10.15774/PPKE.ITK.2014.001

4.4.1.3 Greedy partitioning based on spatial information

The second step of the procedure is a greedy clustering method, which creates rectangular clusters based on the spatial information of the vertices. The height of the rectangular domains can be chosen arbitrary, however, in the demonstrated CFD example it was set to two. The clustering starts from the top left corner, and the largest possible rectangular cluster is created which still meets the I/O constraint (defined in Problem 1). Next, the algorithm moves right and the rectangular-based clustering is continued on the unclustered vertices. If there are no more unclustered vertices in the selected layers the algorithm moves down and continues with the lower layers.

In spite of the greedy nature of the clustering method, the decisions are made on comprehensive information, as the vertices have been already positioned in such a way that short interconnections will enforce locally coupled clusters during greedy clustering. The resulting partitioning is shown in Figure 4.6.

4.4.2 The configuration of the hMetis program

Unfortunately, the F_{Area} objective used in Problem 1 cannot be selected in the program, therefore, a similar metric, the *sum of external degrees* (SOED), was chosen. The *subdomain degree* of partition class V_i is equal to the sum of the weights of the hyperedges that contain at least one vertex in V_i and one vertex from outside of V_i:

$$f_D(V_i, P, G) := \sum_{e \in I(e,V_i)} w(e) \tag{4.14}$$

where $I(e, V_i) = \{a \in E : f_{control}(a, V_i, P, G) \neq 0\}$ and $w(e)$ is the weight of an edge e. Sum of external degree of a partition is the sum of the subdomain degrees:

$$F_{SOED}(P, G) = \sum_{V_i \in V} f_D(V_i, P, G) \tag{4.15}$$

In a hypergraph with edges of unit weight, F_{SOED} can be related to F_{Area} by the following lemma:

Lemma 1

$$F_{SOED}(P, G) = F_{Area}(P, G) + F_{MinCut}(P, G)$$

where $F_{MinCut}(P, G)$ is the standard metric measuring the number of cut hyperedges.

DOI:10.15774/PPKE.ITK.2014.001

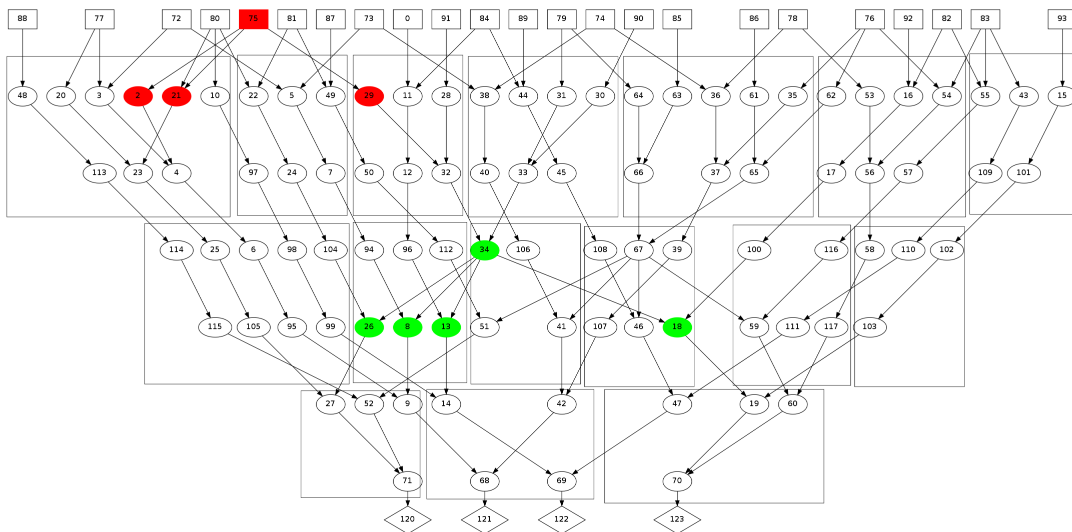

Figure 4.6: Partition of the structured CFD graph created by the proposed algorithm. Inputs and outputs of the mathematical expression are represented by small rectangles and diamonds, respectively, while floating-point units are represented by circles. The clusters of the partitioning are indicated by the transparent rectangles containing vertices. Vertical and horizontal positions of the vertices are determined in the first phase of the algorithm to minimize the distance between the vertices. Given a smart placement of the vertices, a simple greedy mechanism can generate locally coupled clusters which observe the I/O constraint needed for high-performance operation. The figure illustrates how the data-flow graph representing the arithmetic unit looks like after placement and partitioning. The key advantage of the proposed technique is that, during implementation, the clusters can be easily mapped to the FPGA keeping the same relative positions as shown in Figure 4.7. Vertices coloured by red and green are typical examples for vertices which should be placed close to one another because nets with larger fan-out and distant terminals are less likely to be routed efficiently.

DOI:10.15774/PPKE.ITK.2014.001

Proof 1 *Basically, the three metrics differ only in how a cut hyperedge e is counted. Let assume that a hyperedge e has vertices in k different partition classes. SOED, Area, and MinCut objectives measure e with k, $k-1$, and 1, respectively.*

hMetis was instructed to compute direct k-way partitioning (FM) using the F_{SOED} cost function and balancing factor 10. (Balancing factor is a parameter used to specify the allowed imbalance between the size of the partition classes.)

During operation, vertices corresponding to global input and output were fixed and only the rest was partitioned. As a consequence of Lemma 1, F_{SOED} efficiently minimized the area requirements, however, the program could not handle the I/O constraint defined on each class. To overcome the limitations, the program was placed in a framework, where it was executed several times with different seed parameters. Finally, the best candidate, in which the I/O connections of the classes were balanced and close to the desired limit, was selected by hand.

4.4.3 Comparison and evaluation

Both partitioning algorithms have been tested in case of the structured CFD problem. The generated VHDL descriptions of the AUs were implemented on a Xilinx Virtex-6 SXT FPGA (XC6VSX315T) with the standard Xilinx tools at speed grade -1. The algorithms were compared based on the maximal operating frequency reached by the place-and-route process.

This stage of my research can be regarded as an empirical study, which is focusing on the practical implementation of the circuit, to validate the proposed optimization problem (see Problem 1) and to give a greedy algorithm to demonstrate the improved operating frequency of the partitioned AU. In this conceptual study, possible deadlocks or the overall latency generated by the partitioning have not been considered, however, they are all addressed and solved in a more complex algorithm presented in Section 4.5. The solution of these problems requires only a few adjustments in the objective functions, which do not alter the concepts validated in this section, but make the design methodology applicable in real-world applications.

In the beginning of the investigation, I examined simpler data-flow graphs (e.g. FIR filter, CNN state equation) in which case it was possible to manually sort out the deadlock-free partitions from the generated results. In case of more complex data-flow

DOI:10.15774/PPKE.ITK.2014.001

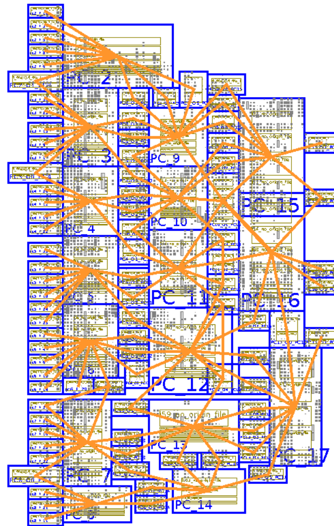

Figure 4.7: Placement constraints (blue rectangles) and placed instances (grey areas) of the arithmetic unit of the CFD problem. Orange lines indicates the connections of the different parts of the circuit. The figure was generated with the standard Xilinx place-and-route editor. In the editor I manually drew a placement constraint for each FIFO and each cluster using the results of the greedy technique. The circuit elements were placed by the Xilinx place-and-route tool keeping my constraints valid. Although the presented solution has more clusters and cut edges than the results of other partitioning algorithms, its key advantage is that the mapping of clusters into the FPGA is straightforward and can be automated using the spatial position of the vertices. (To observe the similarity, rotate Figure 4.6 by $-90°$.)

graphs, the manual sorting turned out to be impossible. Although neither the proposed nor the reference algorithm cannot guarantee a deadlock-free practical solution, the comparison of their results demonstrated the key requirements for high frequency operation in FPGA. With the greedy algorithm, I empirically showed that placement-aware partitioning can lead to a higher operating frequency and common graph partitioning algorithms can only partially address the placement objectives.

The arithmetic unit of the CFD problem consists of nearly 50 floating-point units, which were partitioned into 17 locally controlled partition classes. Separate placement constraint blocks (called *pblocks*) were created for each class and each synchronization FIFO. According to my experiments, Xilinx place-and-route is likely to disperse the registers of the FIFOs without placement constraints, however, floating-point units are placed efficiently even without them.

The pblocks were placed manually with the help of a graphviz [60] plot depicting

DOI:10.15774/PPKE.ITK.2014.001

Table 4.1: Implementation results of different partitioning strategies in case of the 32 bit structured CFD problem.

	No partitioning	hMetis	Proposed algorithm*
Number of clusters	-	7	16
Number of extra FIFOs	23	49	89
Number of Slice Registers	15,534	18,866	21,998
Number of Slice LUTs	12,084	14,275	16,883
Number of occupied Slices	4,039	4,284	5,751
Clock frequency (MHz)	293.97	325.627	369.959

* with manual placement constraints

Table 4.2: Comparing operating frequency of the 32 bit and the 64 bit AU in case of the structured CFD problem. During placement no manual tuning was performed.

	hMetis	Proposed algorithm
32 bit	325.627 MHz	327.761 MHz
64 bit	301.205 MHz	295.596 MHz
difference	8.1%	10.8%

the connections between the clusters. In case of the proposed algorithm, the placement was very straightforward and the resulting layout is shown in Figure 4.7. In case of the reference algorithm, the placement of clusters to minimize the length of their connection was very challenging, and the operating frequency could be improved only slightly by manual tuning. In the configuration one pblock was created for each global I/O FIFO and one for the rest of the circuit.

Partitioning results, resource utilization and operating frequency are compared on Table 4.1 in case of the 32 bit structured CFD problem. The proposed algorithm has reached 13% improvement in operating frequency compared to the naive partitioning, and approximately 25% compared to the unpartitioned case. The price of the improvement was a nearly 34% increase of the area requirements, which is acceptable in a high-performance application.

As a consequence of probabilistic heuristics used in the Xilinx's place-and-route algorithm, the operating frequency of the final circuit is sensitive to the input seed parameters. During the performance evaluation, the generated circuit was implemented

DOI:10.15774/PPKE.ITK.2014.001

with wide range of seed parameters and the highest frequency was selected.

To be able to compare the greedy algorithm with the algorithm described in Section 4.5, the measurements were repeated for 64 bit case and without manual tuning of placement constraints (see Table 4.2). As 64 bit FPUs have larger size and the data-path is also doubled, the increased area requirement of the 64 bit circuit made the interconnecting signals longer resulting in a decreased operating frequency (approximately 8-11%).

DOI:10.15774/PPKE.ITK.2014.001

4.5 Partitioning and placement together

Inspired by the success of the greedy algorithm presented in Section 4.4, a more complex algorithm has been designed to address all the side effects of the partitioning presented in Section 4.2. The algorithm solves fundamentally the same optimization problem (see Problem 1) as the greedy one, however, some extra objective functions are added to the optimization and the idea of placement aware partitioning is also developed further. The proposed design methodology can be regarded as a high-level circuit partitioning strategy, in which the high-level placement (called *floorplan*) and partitioning objectives are combined together to create a special partitioning of the FPUs, where the FPUs are positioned and the clusters are locally coupled.

In Section 4.5.1, the necessary properties of the partitions that can be efficiently mapped into FPGAs are enumerated and discussed. In Section 4.5.2, the proposed algorithm, which is capable of handling all the objectives, is presented. In Section 4.5.3, a framework is described, which was designed to test the partitioning algorithms and to automate the mapping process of mathematical expressions. Finally, in Section 4.5.4, the results of the new algorithm are presented including an experimental analysis of how circuit performance is affected by the maximal number of the I/O connections of the clusters.

4.5.1 Properties of a good partition

The primary aim of the partitioning of the data-flow graph is to find such a partitioning of the FPUs where the clusters, after being implemented in FPGA, are only connected locally. Unfortunately, partitioning can lead to several side effects (see Section 4.1), which shall be addressed in a *good partitioning* to create a high-performance AU applicable in practice. Hereby, the properties of a good partition are enumerated and explained.

1. **The number of I/O connections of each clusters is bounded by a user defined constant.**
   To implement a fast CU the complexity of the CU has to be decreased, which depends on the number of I/O connections. See the definition of $F_{Control}$ in Equation 4.4.

DOI:10.15774/PPKE.ITK.2014.001

2. **The number of cut arcs is minimal.**
   Every cut arc is replaced by a synchronization FIFO, which increases the area requirement of the circuit. See the definition of F_{Area} in Equation 4.2.

3. **The input arcs of each cluster are roughly on the same pipeline level.**
   Clusters which have input arcs on different pipeline levels require larger synchronization FIFOs to guarantee the continuous operation and can increase the overall pipeline length of the AU.

4. **There is no directed cycle in the cluster adjacency graph.**
   Mutually dependent clusters never start to read input data and cause a deadlock in the AU.

5. **The clusters can be mapped to the FPGA without long interconnection between the clusters.**
   Mapping elements of a circuit description into FPGA is a 2D placement problem, where routing resources are limited. In the implemented circuit high fanout and long interconnections should be avoided, otherwise they limit the operating frequency of the whole circuit. To reach significant speedup in the operating frequency of the AU, both the partitioning problem and the placement of the clusters should be solved. The operating frequency could be further increased if the placement of the clusters were explicitly set by using pblocks, however, the partitioning should provide significant speedup even without the physical constraints.

In the proposed algorithm, both steps (placement and partitioning) of the greedy algorithm have been improved and replaced by simulated annealing. For the partitioning step, a new representation has been designed, in which a set of objective functions can be easily defined to target the properties of a good partition. Instead of maximizing the speedup by using physical constraints, my motivation is to investigate how the described properties of the circuit and the free parameters of my algorithm affect the performance of the AU.

DOI:10.15774/PPKE.ITK.2014.001

4.5.2 The proposed algorithm

The main idea of the algorithm is to combine the partitioning and the placement objectives in a two-step procedure. In the first step, an initial and simplified floorplan of the FPUs is created with simulated annealing to minimize the distance between the connected FPUs. In the second step, the floorplanned FPUs are partitioned by another simulated annealing to find a good partition with the previously described properties. The resulting clusters can be easily placed on the FPGA and the lack of long interconnections results in a high operating frequency.

4.5.2.1 Preprocessing and Layering

The same preprocessing and layering are applied as in the greedy algorithm (see Section 4.4.1.1). The input is a mathematical expression described in a text file and the output is a layered graph (see Definition 3), where all vertices have an initial spatial position.

4.5.2.2 Floorplan with simulated annealing

During the floorplan, vertices are horizontally positioned to minimize the length of the interconnections and to prepare the partitioning phase.

In the framework, a simplified homogeneous floorplan is used where every vertex has a unit width, however, the principles used during partitioning can be adapted to floorplans where the size of the different resource types are distinguished. The blocks in one layer are represented by their sequence which is the 1D version of the famous sequence pair representation [31]. This representation is appropriate for ASIC floorplanning, however, in our case, empty spaces can be favorable inside the design, therefore place holder (*bubble*) vertices have been introduced. To distinguish the bubble vertices, they are indicated by negative indices. To limit the complexity of the problem, the number of the bubble vertices on a layer is limited.

The pseudocode of the floorplan is summarized in Algorithm 2. Before the simulated annealing starts, the floorplan is initialized at the preprocessing phase by the Barycentre heuristic [57]. During the simulated annealing, in each iteration, the sequence of the vertices of a random layer is perturbed. Layers are selected with probability proportional to the number of vertices (N_l) they contain. In the selected layer (l),

DOI:10.15774/PPKE.ITK.2014.001

a vertex is selected with probability $(1/(N_l+1))$ or a bubble is created with probability $(1/(N_l+1))$. If a normal vertex is selected, it will be swapped with one of its neighbor. If a bubble vertex is selected, its size will be increased or decreased by one. If a bubble with size one is selected for decreasing, it will be deleted.

Algorithm 2 Pseudocode of the simulated annealing used for floorplanning

Require: Layered graph with pre-positioned vertices
1: **while** the stopping condition is not reached **do**
2: Randomly select a layer with probability proportional to its size.
3: Compute the cost related to the selected layer.
4: Perturb the layer by swapping two vertices, or create/increase, or decrease/delete a bubble vertex
5: Compute the new cost related to the selected layer.
6: Accept or reject the perturbation based on the cost difference and the simulated temperature.
7: Update the stopping condition and the simulation temperature.

The linear combination of the following three objective functions is minimized during the simulated annealing:

1. **Total squared distance (TSD) of the connected vertices**
   This objective minimizes the distance between the connected vertices. As the clusters are determined based on the position of the vertices, this objective automatically avoids long interconnections between the clusters. Distance between two vertices are determined according to their horizontal coordinates:

$$distance(A,B) := \begin{cases} (x_A - x_B)^2 & \text{if A and B are connected} \\ 0 & \text{otherwise} \end{cases}$$

   where x_A and x_B are the horizontal coordinates of vertex A and B, respectively. Vertical coordinates can be neglected because the distance is always one as the graph is layered.

2. **Maximum distance (MDV) between connected vertices**
   This objective is used beside TSD to put an extra pressure on the longest interconnection because usually the longest interconnection has the largest delay limiting the operating frequency.

DOI:10.15774/PPKE.ITK.2014.001

3. **Maximum distance (MDI) between vertices which get input from a common vertex**
   If the fanout of the output data signal of a vertex is larger because the vertex supplies data to several other vertices, it is practical to put the target vertices close to one another. In this case, the fanout of the data signal can be tolerable and the partitioning phase can put the target vertices into the same cluster to decrease the number of the cut arcs.

The result of the simulated annealing in case of the presented CFD problem is shown in Figure 4.9. In our experiments, the following coefficients were used in the global objective function: TSD=0.2, MDV=3, MDI=6. To get a practical solution, usually, 1K-10K iterations have been executed.

4.5.2.3 New representation for graph partitioning

The input of the partitioning is a layered graph where the horizontal coordinates of the vertices are already set. To force horizontal cutting, the user can group the layers into *belts* before the partitioning starts and partitions on each belt are represented separately. The height of the layers should be set according to the complexity and the pipeline length of the given mathematical expression. In our experiments, the belts have been 2 or 3 layers high.

The main idea of the proposed representation is that vertices inherit their affiliation (cluster ID) from a neighboring vertex or are assigned to a new cluster. Clusters of partitions that can be represented this way automatically form continuous non-overlapping regions. If vertices are already connected with short interconnections, a new simulated annealing can be used to find a partition, where clusters are continuous, non-overlapping and only connected to the neighboring clusters (*locally connected*). To reach local connectivity, the length of the connections between the vertices shall be shorter than the width of the clusters.

In our case, the vertices have uniform size and the direction of the inheritance can be described with a *spin* associated to every vertex. In case of variable size vertices, spins cannot describe all the possible partitions which have continuous clusters, therefore, other descriptors should be used beside the spins (e.g. visiting order of the vertices). The possible spin values are the following:

DOI:10.15774/PPKE.ITK.2014.001

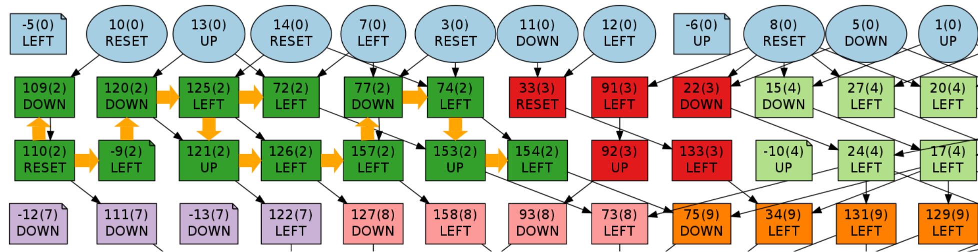

Figure 4.8: A fragment of the first belt of Figure 4.9 is shown to demonstrate how inheritance works.

- LEFT : Vertex will inherit cluster ID from the left neighbor.
- DOWN : Vertex will inherit cluster ID from the bottom neighbor.
- UP : Vertex will inherit cluster ID from the upper neighbor.
- RESET : Vertex will be assigned to a new cluster.

For an example how inheritance works, see thick arrows and cluster 2 in Figure 4.8.

The vertices of a belt are assigned to *columns* based on their horizontal position. Two vertices are assigned to the same column if and only if they have the same horizontal position. In a worst-case situation, the number of the columns in a belt is equal to the number of the vertices of the belt.

When a partition is built up from a representation (see Algorithm 3), the columns of the given belt are visited from left to right. In each step, all the vertices of a column are assigned to clusters. First, the vertices of the given column are visited from top to down and each vertex inherits its cluster ID according to its spin. If the inheritance is ambiguous, the vertex is not clustered. If unclustered vertices remain in the column, the vertices are revisited in bottom-up order. During the second visit of a vertex, the IDs are also associated based on the spins, however, if the inheritance is ambiguous, the vertex is associated to a new cluster. In worst case all the vertices are visited twice, therefore the partition can be built up in $O(N)$ steps, where N is the number of the vertices on the given belt.

The representation is implemented via the following data structures. To associate each cluster with a cluster ID and a spin, the map container from the C++ STL library can be used. The column-based visiting order of the vertices can be realized with

DOI:10.15774/PPKE.ITK.2014.001

Algorithm 3 Build the partition described by the proposed representation

Require: Each vertex is associated with a spin.
Require: Vertices grouped into columns based on horizontal position.
Require: Spatial neighbors of each vertex is stored.
1: **for** each column c **do**
2:     **for** each vertex v of c from *top* to *down* **do**
3:         Assign cluster ID to v according to its spin, if it is possible.
4:     **for** each unclustered vertex v of c from *down* to *top* **do**
5:         Assign cluster ID to v according to its spin, if it is possible.
6:         **if** v is unclustered **then**
7:             Assign an unique cluster ID to v.

nested vectors. The spatial neighbors of the vertices can be described with a 2D array, called the *neighboring array*, which size equals to the number of vertices multiplied by the possible spin directions. Although regular vertices have unit width, the size of bubble vertices are different. Thus the neighboring information cannot be concluded solely from the column information. Before the iteration starts, the neighboring array has to be computed based on the position and size of the vertices.

The representation could be extended for variable sized vertices in exchange for additional columns in the neighboring array and a more complex visiting procedure. If the size of the verices are different, the maximal number of possible neighbors can be determined based on the ratio of the size of the smallest and the largest vertices. In this case the spins would indicate the number of the neighbor from which the cluster ID is inherited. In a practical implementation, the maximal number of the spatial neighbors of a vertex, that is the ratio of the size of the smallest and the largest vertices, has to be limited, and the neighboring array is allocated according to this limitation. With the variable size vertices, a more realistic model of the floorplan of the floating-point units can be given, however, in my experiments I found it less useful as the exact size of the units has less relevance in high-level floorplans .

One of the main benefits of the layering and the belts is that directed cycles induced by partitioning can only occur inside the belts, which can be eliminated by minimizing the NCNN and the NMD objectives during partitioning as discussed in Section 4.5.2.4.

Lemma 2 *Assuming a layered data-flow graph in which the belts are partitioned separately, if there is any directed cycle in the cluster adjacency graph all the vertices of*

DOI:10.15774/PPKE.ITK.2014.001

the directed cycle must belong to the same belt.

Proof 2 *In a layered data-flow graph (see Definition 3) an arc coming from a vertex of layer i is always directed to a vertex residing in layer $(i+1)$. As belts are distinct groups of consecutive layers, arcs crossing belt borders are always directed to the belt containing the layers with higher IDs. Consequently, in the cluster adjacency graph if a route leaves a belt, it cannot return and cannot form a cycle.*

4.5.2.4 Partitioning

The pseudocode of the partitioning procedure is displayed in Algorithm 4. The initial partition is built up based on random spins associated to every vertex. To avoid meaningless spin directions, the spins of the vertices which are at the belt boundaries are not allowed to direct outward the belt.

In each iteration of the simulated annealing, one belt is selected with a probability proportional to the number of vertices it contains. In the belt one vertex is selected randomly and its spin is perturbed, however, meaningless spin directions are not allowed. In the next step, the partition on the selected belt is rebuilt and the linear combination of the following objective functions is used to compute the energy function of the simulated annealing.

1. **Total number of cut arcs (TNC)**
   According to the properties of a good partition, the number of cut arcs should be minimized. See the definition of F_{Area} in Equation 4.2.

2. **Total control penalty of the clusters (TCP)**
   The *control penalty* of a cluster V_i is defined as

$$F_{CP}(V_i) = \begin{cases} F_{Control}(V_i) - T_{user} & \text{if } F_{Control}(V_i) > T_{user} \\ 0 & \text{otherwise} \end{cases}$$

   where $F_{Control}$ was defined in Equation 4.4 and T_{user} is the user defined threshold to limit the controlling cost (also called as I/O cost) of each cluster.

3. **Number of clusters (NC)**

DOI:10.15774/PPKE.ITK.2014.001

4. **Number of connections between non-neighboring clusters which are on the same belt (NCNN)**
   To find a partition in which clusters are only connected to their neighbors NCNN should be zero.

5. **Number of mutual dependencies between neighboring clusters which are on the same belt (NMD)**
   According to Lemma 2, partitioning can only introduce cycles (causing deadlock) inside the belts. On the other hand, if NCNN is minimized to zero, only neighboring clusters can be connected inside the belts. Therefore, cycles can only exits via the connections of neighboring clusters, if they mutually depend on each other. I introduced the NMD objectives to count the mutual dependencies between the neighboring clusters and if both NCNN and NMD are zero then no directed cycle can be present in the belts.

If the new partition is not accepted, the perturbed spin is reverted and the partition on the selected belt is rebuilt.

Algorithm 4 Pseudocode of the simulated annealing used for partitioning

Require: Layered graph with positioned vertices
1: Set the spin of each vertex randomly, but avoid meaningless directions.
2: **while** the stopping condition is not reached **do**
3: Compute the partitioning cost.
4: Randomly select a layer with probability proportional to its size.
5: Perturb the layer by altering the spin of a randomly selected vertex.
6: Rebuild the partitioning of the belt affected by the perturbation.
7: Compute the new partitioning cost.
8: Accept or reject the perturbation based on the cost difference and the simulated temperature.
9: Update the stopping condition and the simulation temperature.

The result of the partitioning is shown in Figure 4.9. In our experiments the following coefficients were used to compute the energy function: $C_{TNC} = 1$, $C_{TCP} = 5$, $C_{NC} = 2$, $C_{NCNN} = 18$, $C_{MDI} = 10$. To get a practical solution, usually, 10K-100K iterations have been executed.

DOI:10.15774/PPKE.ITK.2014.001

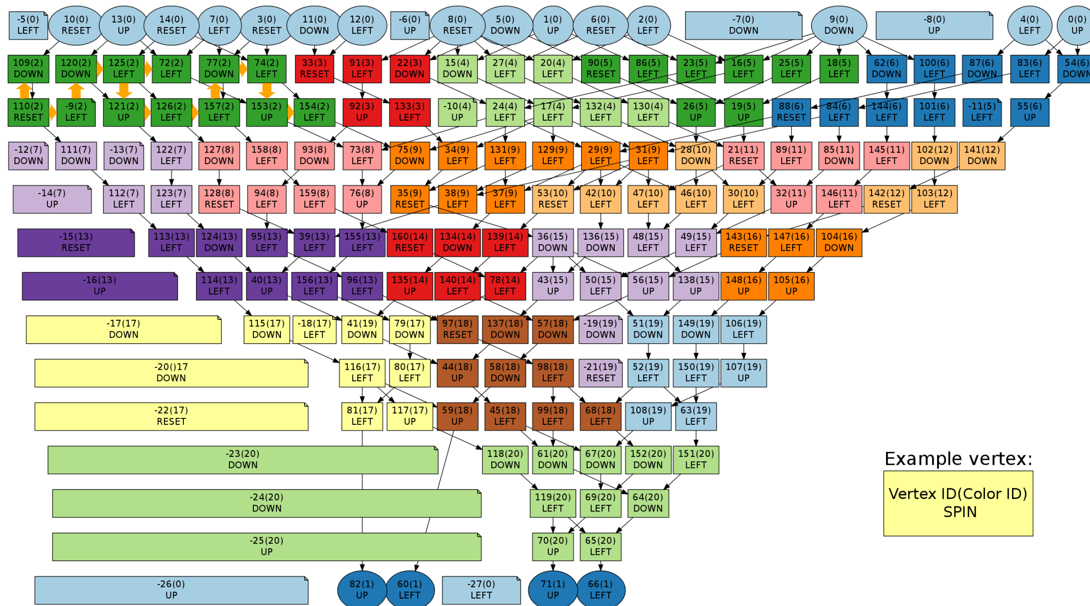

Figure 4.9: The partitioned data-flow graph generated from the numerical scheme of the unstructured CFD problem. Bubble vertices are indicated by negative indices.

4.5.2.5 Outline of the full algorithm

The outline of the full algorithm is summarized in Algorithm 5. The input of the algorithm is a mathematical expression which has to be evaluated by the arithmetic unit. After the data-flow graph has been constructed, the graph is layered and each vertex is associated with an initial coordinate the same way as in the greedy algorithm. Next, the vertical coordinates of the vertices are finalized via the floorplan presented in Section 4.5.2.2. To guarantee continuous and non-overlapping partitioning of the positioned vertices, I proposed a new representation in Section 4.5.2.3. The final partition is created via a simulated annealing presented in Section 4.5.2.4 utilizing the new representation.

4.5.2.6 Comparison to the terminal propagation technique

The terminal propagation technique described in Section 4.3.4.1 can be regarded as an alternative technique to combine placement objectives into partitioning. Although it assumes a priori knowledge of the number and the topology of clusters, which is an obvious limitation compared to my solution, it may be extended to produce partitions

DOI:10.15774/PPKE.ITK.2014.001

Algorithm 5 Pseudocode of the full algorithm

Require: A mathematical formula described in a text file
1: Parse the mathematical formula and form the data-flow graph.
2: Perform layering on the graph; every vertex is associated with a vertical coordinate. (Section 4.4.1.1)
3: Initialize the horizontal coordinates of the vertices via the Barycentre heuristic. (Section 4.4.1.2)
4: Determine the final horizontal coordinates of the vertices via the simulated annealing described in Section 4.5.2.2.
5: Form the new graph representation presented in Section 4.5.2.3.
6: Using the new representation and another simulated annealing get the final partitioning. (Section 4.5.2.4)

similar to ones presented in the dissertation. One can design a procedure, in which the developer first describe a topology similar to the one presented in Figure 4.9, and then the graph is recursively bi-partitioned to form clusters with the required topology. To force the local connectivity of the clusters, the edges connecting the virtual vertices with the unpartitioned vertices can be even weighted based on the distance in the topology.

Due to the recursive partitioning nature of the terminal propagation, it has further limitations. Compared to the simulated annealing, it can be regarded as a greedy approach, as the successive partitionings cannot improve the cluster boundaries of the previous partitionings. Furthermore, complex objective functions cannot be precisely evaluated until the last partitioning. During the previous partitionings, objectives and constraints can only be estimated which makes the sharp approximation of constraints like in Problem 1 very difficult.

Contrary to the limitations of the terminal propagation technique, similar characteristics can also be observed. The belts defined in the proposed algorithm can be regarded as a special constraint on the topology of the clusters and are related to the fixed topology used in terminal propagation. Furthermore, the movement of vertices toward their neighbors during floorplan is similar to the effect of the virtual vertices which are introduced to attract the unpartitioned vertices toward their partitioned neighbors.

DOI:10.15774/PPKE.ITK.2014.001

4.5.3 Framework

A framework has been implemented in C++ to automatize the generation of the AU from a textual or a SystemC [61] description of the numerical scheme. In the framework, the scheme is represented via a hypergraph, on which the proposed partitioning algorithms can be tested. In case of SystemC, which is itself a C++ library for high-level circuit description, only an extra header file has to be included in the project to generate the hypergraph representation. The developed hypergraph library is inspired by the Lemon graph library [62]: the hypergraph is stored internally via its sparse incidence matrix and simple STL-like iterators are implemented for arc, vertex, source, target or neighborhood traversals. The modification of the original library was necessary to precisely represent high-level circuits.

The operators in the partitioned data-flow graph and the synchronization FIFOs are implemented via Xilinx IP Cores [63]. The connecting signals, the control logic and the mixer units, which supply the AU with data, are developed in VHDL with the Xilinx ISE Design Suite 13.1 [64]. The placement constraints are defined in the Xilinx's user constraint file (UCF) and submitted at the place-and-route phase. If manual tuning of placement constraints is chosen, the Xilinx PlanAhead [65] editor is used. Finally, the generated AUs were implemented on a Xilinx Virtex-6 SX475T FPGA with speed grade -1 (see Section 2.1 for more information).

4.5.4 Results

Using the VHDL generation feature of the framework, the effect of the T_{user} parameter of the partitioning, which limits the control cost of a cluster, has been investigated in case of two CFD problems presented in Section 3.1.2. In both cases the unpartitioned version and the fully partitioned version of the AU have been implemented as references. In the unpartitioned version, one CU is assigned to the whole circuit, while in the fully partitioned version each FPU is controlled by a separate CU. The resulting operating frequencies and area requirements are summarized in Figure 4.10, while partitioning and implementation details are in Table 4.3 and 4.4.

In the structured CFD example, two interfaces can be computed in the AU requiring 23 input variables, 4 output variables and 44 FPUs (14 multipliers and 30 adders). The large number of I/Os of the AU results in slow operating frequency in the unpartitioned

DOI:10.15774/PPKE.ITK.2014.001

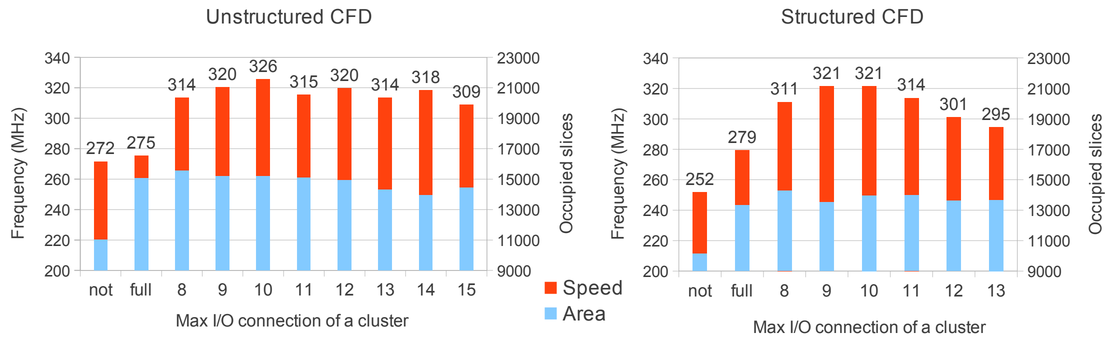

Figure 4.10: Operating frequency and area requirements of the AU as the maximum number of I/O connections of a cluster is changing.

case, however, if the I/Os of the clusters are limited to 9 or 10, the operating frequency can be improved by 15-27%.

In the presented unstructured CFD example, only one interface is computed in the AU, requiring 15 inputs, 4 outputs and 64 FPUs (30 multipliers, 25 adders, 9 specials). Special FPUs include FPUs which are used for negation or division by 2. The latter one is implemented by modifying the exponent of the floating point numbers. Similarly to the previous example, the largest speedup (18-19%) can be reached if the number of I/O connections of the clusters is limited to 10, although clusters with slightly more I/O connections can also run at high operating frequency.

In the measurements, the maximal frequency was determined via the same parameter sweep technique, which was described at the greedy algorithm in Section 4.4.1. During place-and-route phase, only one pblock was created for the whole AU, and no manual tuning was employed. All the generated circuits are dead-lock free, and the increase of the overall pipeline length (latency) of the AUs is approximately $20-30\%$, which is tolerable in high-performance applications. Compared to the greedy algorithm, the operating frequency was increased by approximately 8%.

4.6 Summary

In the chapter a new high-level technique was proposed to design high-performance arithmetic units for FPGA. The presented scientific results form the basis of my first

DOI:10.15774/PPKE.ITK.2014.001

Table 4.3: Partitioning and implementation results of the structured CFD graph.

				Cluster I/O threshold (T_{user})					
		Unpartitioned	Fully-partitioned	8	9	10	11	12	13
Num. of clusters		1	44	19	100	128	143	100	103
Extra delay vertex		26	0	33	32	32	37	28	31
Pipeline length		100	100	128	100	128	143	100	103
Max cluster cut		27	6	8	9	10	11	12	13
Cut arcs	Outside	27	46	39	38	38	35	35	35
	Inside	0	46	56	56	52	52	49	49
	Total	27	92	95	94	90	87	84	84
FIFO	32	23	68	63	72	60	47	65	60
	64	0	18	26	22	27	36	19	24
	128	4	6	6	0	3	4	0	0
	TOTAL	27	92	95	94	90	87	84	84
Area	FF	41,664	48,591	51,544	51,137	50,775	50,787	49,641	49,783
	LUT	31,384	37,297	40,022	38,337	38,869	39,387	36,583	37,397
	DSP	244							
Frequency (MHz)		251,889	279,33	311,139	321,44	321,44	313,775	301,114	294,638
Improvement		100%	110,89%	123,52%	127,61%	127,61%	124,57%	119,54%	116,97%
			100%	111,39%	115,08%	115,08%	112,33%	107,80%	105,48%

thesis group.

I designed a new local control for arithmetic units generated from complex mathematical expressions to avoid global control signals, which are frequently the bottleneck of the overall performance (Thesis I.1). The floating-point units of the arithmetic unit are partitioned into independently controlled clusters, which are synchronized via FIFO buffers. The new feature of the proposed control is that instead of the floating-point units only the input and the output FIFOs of the clusters are controlled. I proposed a constrained optimization problem to find a partitioning of the arithmetic unit, in which the number of synchronizing FIFOs is minimized while the number of I/Os controlled by each control unit is constrained.

I created a new greedy algorithm for arithmetic unit partitioning to demonstrate that locally coupled clusters with the proposed local control can outperform the unpartitioned arithmetic units or the arithmetic units partitioned with standard algorithms (Thesis I.2). The new feature of the algorithm is that the partitioning is carried out on a high-level floorplan of the floating-points units to promote the local connectivity of the clusters. I empirically showed that during FPGA implementation the maximal operating frequency depends on both the complexity of the control units and the topology of clusters, therefore, naive partitioning algorithms, which cannot handle placement objectives, produce less competitive results.

DOI:10.15774/PPKE.ITK.2014.001

Table 4.4: Partitioning and implementation results of the unstructured CFD graph.

				Cluster I/O threshold (T_{user})							
		NP*	FP*	8	9	10	11	12	13	14	15
Num. of clusters		1	64	32	26	21	20	19	17	14	13
Extra delay vertex		37	0	43	46	50	49	48	48	50	51
Pipeline length		165	165	179	198	201	199	199	183	183	197
Cut arcs	Outside	19	44	26	25	25	24	25	21	22	20
	Inside	0	79	88	79	76	74	72	68	63	64
	Total	19	123	114	104	101	98	97	89	85	84
Max cluster cut		27	6	8	9	10	11	12	13	14	15
FIFO	32	15	101	82	71	67	60	66	66	63	55
	64	0	12	29	30	30	31	27	23	22	26
	128	2	6	3	3	4	7	4	0	0	3
	256	2	4	0	0	0	0	0	0	0	0
	TOTAL	19	123	114	104	101	98	97	89	85	84
Area	FF	43,966	55,606	58,052	56,980	56,830	56,393	56,139	54,977	54,477	54,566
	LUT	33,043	43,035	42,950	42,419	42,254	42,615	41,723	39,846	39,563	40,126
	DSP	405									
Frequency (MHz)		271,655	275,331	313,513	320,307	325,627	315,457	319,591	313,578	318,407	308,833
Improvement		100%	101,35%	115,41%	117,91%	119,87%	116,12%	117,65%	115,43%	117,21%	113,69%
			100%	113,87%	116,34%	118,27%	114,57%	116,08%	113,89%	115,65%	112,17%

* where NP = not partitioned and FP = fully-partitioned.

Using the ideas demonstrated with the greedy algorithm, I developed a more complex partitioning algorithm, which can generate high-performance deadlock-free arithmetic units with moderate pipeline lengths (Thesis I.3). The new algorithm is similar to the greedy one as both require an initial floorplan of the floating-point units, however, it has several new features, which make it superior and applicable in real world problems. One of the most important feature is that instead of the greedy mechanism, it is based on simulated annealing, which can handle a more complex objective function, with which mutual dependencies of the clusters can be eliminated. Furthermore, it contains a new representation of the floorplanned vertices, which can narrow the search space of the simulated annealing to the continuous and non overlapping clusters. The algorithm has been demonstrated in case of two complex CFD problems outperforming the unpartitioned case by 15-25%.

The primary application of the demonstrated technique is the acceleration of CFD simulations via the Falcon architecture, however, as the Falcon architecture can be adapted to other partial differential equations, the technique can be used for the acceleration of other type of simulations, e.g. seismic waves or electromagnetic fields.

The Virtex-7 FPGA familiy, which is already available on the market, has approximately 1.5 times more physical resources than the Virtex-6 family, while its operating

DOI:10.15774/PPKE.ITK.2014.001

frequency remained in the same order of magnitude. By transferring the design to a Virtex-7 FPGA, performance improvement can be realized if more Falcon processors are implemented and connected into a chain. In this case roughly 1.5 times more performance can be expected, although the overall pipeline length of the multi-processor configuration will also increase. The generation after the Virtex-7 family is called Virtex UltraScale, and it is manufactured at the 20 nm technology. It has roughly 1.5 times more physical resources than the Virtex-7 family, which predicts further performance gain for the Falcon implementations. An important characteristic of the new generation is that global clocking network was replaced by local networks forcing the developers to design ASIC-like independent clock regions.

DOI:10.15774/PPKE.ITK.2014.001

Chapter 5

Density Matrix Renormalization Group Algorithm

The DMRG is a variational numerical approach developed to treat low-dimensional interacting many-body quantum systems efficiently [10, 66, 67]. In fact, it has become an exceptionally successful method to study the low energy physics of strongly correlated quantum systems which exhibit chain-like entanglement structure [11].

In Section 5.1, previous DMRG implementations and acceleration efforts are reviewed. Section 5.2 describes the models which are used as test cases to demonstrate the operation of the algorithm. Symmetries, which can be exploited to decrease the computational requirements of the algorithm, and the algorithm itself are presented in Sections 5.3 and 5.4, respectively. Preparing for the acceleration of the algorithm, its run-time analysis is given in Section 5.5. Finally, limits of an FPGA implementation are described in Section 5.6. Descriptions of the models, the algorithm, and the symmetries are based on the literature, while the run-time analysis and the FPGA estimation are my own work providing the foundation for the next chapter.

5.1 Previous implementations

The original DMRG algorithm [10] was introduced in 1992 by Steven R. White and was formulated as a single threaded algorithm. In the past, various works have been carried out to accelerate the DMRG algorithm on shared [68] [69] and distributed memory [70–73] architectures, however, none of them took advantage of recent paral-

DOI:10.15774/PPKE.ITK.2014.001

lel architectures: graphical processing unit (GPU) and field-programmable gate array (FPGA).

One of the first parallelizations was [68] converting the projection operation to matrix-matrix multiplications and accelerating them via OpenMP interface. In [72] a similar approach was presented for distributed memory environment (up-to 1024 cores) optimizing the communication between the cores, while in [73] the acceleration of the computation of correlation function had been investigated. Recently, [69] presented an acceleration on shared memory architectures exploiting SU(2) symmetries, while [74] proposed a novel direction for paralellization via a modification of the original serial DMRG algorithm.

The GPU architecture has been successfully employed in neighboring research areas to accelerate matrix operations. In [75] a GPU is used to accelerate tensor contractions in plaquette renormalization states (PRS), which can be regarded as an alternative technique to tensor product states (TPS) or the DMRG algorithm. In [76] the second-order spectral projection (SP2) algorithm has been accelerated, which is an alternative technique to calculate the density matrix via a recursive series of generalized matrix-matrix multiplications.

The work presented in the dissertation is the first attempt (to the best of my knowledge) to investigate how the DMRG method can utilize the enormous computing capabilities of recent parallel architectures (GPU, FPGA). I examined the theoretical performance of both architectures in case of the critical operation of the algorithm. As I found the GPU architecture superior to FPGA, I selected GPU to accelerate the DMRG algorithm. In Section 5.6, the performance limits of a possible FPGA implementation are estimated and discussed, while in Chapter 6, a smart hybrid CPU-GPU acceleration is presented, which tolerates problems exceeding the GPU memory size. Contrary to the previous acceleration attempts, not only the projection operation is accelerated, but further parts of the diagonalization are also computed on the GPU.

5.2 Investigated models

In order to illustrate the underlying features of the algorithm, it is applied to the so-called spin-$\frac{1}{2}$ Heisenberg model and the spin-$\frac{1}{2}$ Hubbard model. The selected models describe how to compute the Hamiltonian of the system of interest, while the main task

DOI:10.15774/PPKE.ITK.2014.001

is to find some of the low-lying eigenvalues and eigenvectors of the Hamiltonian by a diagonalization algorithm. In practice, instead of solving the problem for the complete Hilbert space directly, various physical phenomena can be exploited to reduce the complexity of the problem.

5.2.1 Heisenberg model

The Heisenberg model describes the physics of magnetic systems and provides theoretical description of various experimental measurements. In the model a magnetic system is simulated on a lattice of interacting *spins*. A microscopic magnetic moment (spin) is localized at each lattice site j and described by a quantized, two-valued variable, $\sigma_j \in \{\uparrow, \downarrow\}$, related to the two possible orientations of the spin. Limiting the interactions to only neighboring spins – which is often a good approximation – the Hamiltonian of the model is written as

$$H = \frac{1}{2} \sum_{j=1}^{N-1} \left(S_j^+ S_{j+1}^- + S_j^- S_{j+1}^+ \right) + \Delta \sum_{j=1}^{N-1} S_j^z S_{j+1}^z \tag{5.1}$$

where S_j^+, S_j^- operators change, while S_j^z measures the orientation of the spin on lattice site j. The overall behavior of the system can be tuned via the relevant parameter Δ. The explicit matrix representation of an operator $\mathcal{O}_j$ acting on site j of a chain with N spins is given as

$$\mathcal{O}_j = \bigotimes_{i=1}^{j-1} \mathbb{I} \otimes \mathcal{O} \otimes \bigotimes_{i=j+1}^{N} \mathbb{I} \tag{5.2}$$

where $\mathbb{I}$ is the identity and $\mathcal{O}$ is one of the followings

$$S^+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \; S^- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \; S^z = \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{5.3}$$

The Hamiltonian of N spins acts on the tensor product space of dimension 2^N, that is the dimension of the complete Hilbert space grows exponentially as the size of the system increases.

5.2.2 Hubbard model

The Hubbard model was introduced to describe electrons in solids to characterize the transition between insulating and conducting systems. The single-band Hubbard

DOI:10.15774/PPKE.ITK.2014.001

model provides appropriate description of low temperature systems, where all particles are in the lowest Bloch band and the long-ranged interactions between the particles can be neglected due to strong screening effects [77]. More recently, various multi-band Hubbard models have been applied to high-temperature superconductivity [78] and systems of higher spin to understand the behavior of optically trapped ultracold atoms [79].

In the general spin-F system, each lattice site is characterized by $2F + 1$ two dimensional vectors. Each vector is assigned with a distinct label (from $\{-F, -F + 1, ..., F - 1, F\}$) called spin polarization value (denoted by σ). A vector assigned to a spin polarization σ describes two orthogonal states: the site is occupied ($[0; 1]$) by the particle of spin polarization σ or not ($[1; 0]$). As a consequence, a lattice site of a spin-F system possesses 2^{2F+1} internal degrees of freedom.

The lattice model of interacting particles of spin-F consists of two competing terms: the kinetic term, which describes the tunneling of particles between neighboring lattice sites, and the local potential term, which describes on-site density-density interaction measuring the attraction or repulsion between the interacting particles. The single-band, fermionic Hubbard model of spin-F is defined on a chain with N sites as

$$H = -t \sum_{j=1}^{N-1} \sum_{\sigma=-F}^{F} \left(c^{\dagger}_{j,\sigma} c_{j+1,\sigma} + \text{h.c.} \right) + \frac{U}{2} \sum_{j=1}^{N} \sum_{\sigma \neq \sigma'} n_{j,\sigma} n_{j,\sigma'} \tag{5.4}$$

where t measures the hopping amplitude between neighboring sites and U is the interaction strength. Creation and annihilation operator acting on site j with spin polarization σ, denoted as $c^{\dagger}_{j,\sigma}$ and $c_{j,\sigma}$, adds or removes a particle located on site j with spin polarization σ. The particle density of spin polarization σ on site j is measured by operator $n_{j,\sigma} = c^{\dagger}_{j,\sigma} c_{j,\sigma}$. The explicit matrix representation of an operator $\mathcal{O}_{j,\sigma}$ acting on site j and polarization σ is constructed as

$$\mathcal{O}_{j,\sigma} = \bigotimes_{i=1}^{F'(j-1)} \Phi \otimes \mathcal{O}_{\sigma} \otimes \bigotimes_{i=F'(j+1)}^{F'N} \mathbb{I} \tag{5.5}$$

$$\mathcal{O}_{\sigma} = \bigotimes_{i=-F}^{\sigma-1} \Phi \otimes \mathcal{O} \otimes \bigotimes_{i=\sigma+1}^{F} \mathbb{I} \tag{5.6}$$

$$\Phi = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{5.7}$$

DOI:10.15774/PPKE.ITK.2014.001

where $F' = 2F + 1$, $\mathbb{I}$ is the identity, Φ is the fermionic phase-factor and $\mathbb{O}$ is one of the followings

$$c^\dagger = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \ c = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}. \quad (5.8)$$

The Hamiltonian describing the spin-F system of N lattice sites acts on the tensor product space of dimension $2^{F'N}$, and similarly to the Heisenberg model, the dimension of the complete Hilbert space blows up exponentially. Comparing to the bosonic operators of the Heisenberg model, the key differences in the construction of operators are the appearance of internal quantum number, σ, and the presence of the phase-factor describing the antisymmetric nature of fermionic systems. To ease the comparison of the two models only the $F = \frac{1}{2}$ case is presented, however, the observed tendencies are valid for higher F values.

5.3 Symmetries to be exploited

In many systems the Hamilton operator does not change the value of a measurable quantity, i.e., it commutes with the operator connected to that measurable quantity. These operators are called symmetry operators and can be used to cast the Hilbert space to smaller independent subspaces[80]. Consequently, instead of solving a large matrix eigenvalue problem, the eigenvalue spectrum can be determined by solving several smaller problems. In the Heisenberg model the total spin projection, $S_z = \sum_{j=1}^{N} S_j^z$, is such a symmetry operator. Meanwhile, in the Hubbard model of spin-F the total particle number associated to each spin polarization σ, $N_\sigma = \sum_{j=1}^{N} n_{j,\sigma}$, is conserved. Thus, the distinct quantum numbers helps to partition the Hilbert space into multiple independent subspaces corresponding to a given combination of quantum number values.

A given symmetry operator shares the same eigenvectors of the Hamiltonian, thus the eigenstates of the Hamiltonian can be labeled by the eigenvalues of the symmetry operator (*quantum number*, Q), and the Hilbert space can be decomposed into subspaces (*sectors*) spanned by the eigenvectors of each quantum number value [81]. Introducing a quantum number based representation, the sparse operators (Eqs. 5.2, 5.5) can be decomposed to a set of smaller but dense matrices, furthermore the Hamiltonian operator (Eqs. 5.1, 5.4) becomes blockdiagonal.

DOI:10.15774/PPKE.ITK.2014.001

5.4 Algorithm

The DMRG approach has two phases: in the *infinite-lattice algorithm* the approximated Hilbert space of a finite system of N interacting spins is built up iteratively, while in the optional *finite-lattice algorithm* the number of the interacting spins is fixed and further iterations are carried out to increase the accuracy of the computed results. As in both cases the iterations are very similar, for the sake of simplicity, I consider only the infinite-lattice algorithm. The detailed description of the algorithm can be found in the original work [10] and various reviews [66, 67], here only the key steps of an iteration of the infinite-lattice algorithm are summarized in Algorithm 6. The techniques discussed in the section are not my scientific results, they are part of the DMRG literature and provide the basis of my analysis and acceleration.

Algorithm 6 One iteration of the infinite-lattice algorithm

1: Load a left and a right block and enlarge each block with a site.
2: Form the superblock configuration.
3: Compute the lowest eigenstate of the superblock Hamilton H_{SB}. (Davidson method)
4: **for** each enlarged block **do**
5: Construct the density matrix for the given block from the lowest eigenstate.
6: Compute the eigenvalues of the density matrix. (Lanczos method)
7: Renormalize the basis of the block by keeping states with high eigenvalues.

In the two-site DMRG procedure four subsystems (left block describing l sites, 1 site, 1 site, right block describing r sites) compose the finite system of $N = (l+2+r)$ sites called *superblock*. The sites contained in each block are described maximally by m, optimally chosen states, which can be significantly smaller than the exactly required q^l or q^r basis, where q is the degree of freedom of one site. As the central sites of the superblock are represented exactly by q-q states, the size of the superblock Hilbert space is q^2m^2. Considering, however, the symmetries mentioned above, the problem can be restricted to a subspace of the superblock corresponding to a particular Q value. For example, in case of Heisenberg and Hubbard models, the size of the superblock Hilbert space can be reduced significantly as demonstrated in Figures 5.1 and 5.2, respectively. It is, however, clear that even using symmetry operators the

DOI:10.15774/PPKE.ITK.2014.001

dimension of the reduced space grows exponentially with the size of the lattice (if no truncation is done).

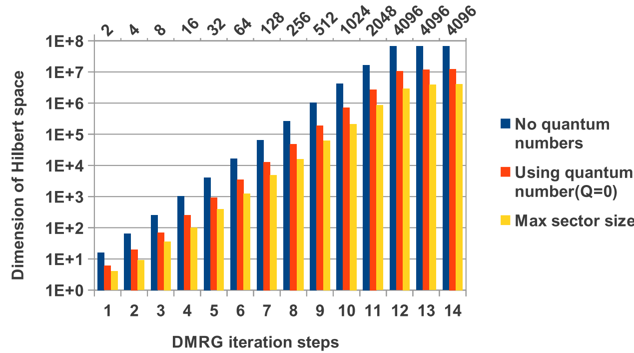

Figure 5.1: Exploiting the projection symmetry in the Heisenberg model, the Hilbert space of the superblock can be restricted to the subspace corresponding to $Q = 0$. The measurements were produced via the CPU-only mode of the implementation presented in Chapter 6. At the top of the chart, labels indicate the number of retained block states (m=4096).

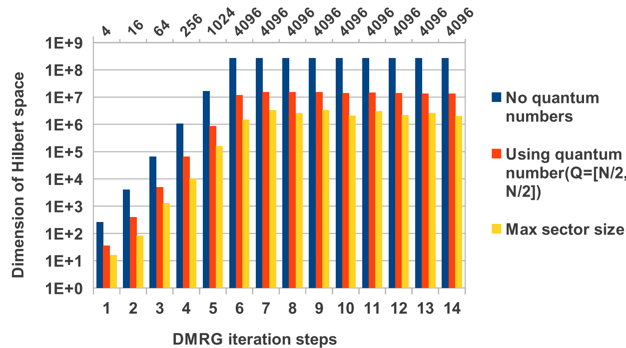

Figure 5.2: Exploiting the conservation of particle number in the spin-$\frac{1}{2}$ Hubbard model, the Hilbert space of the superblock can be restricted to the subspace corresponding to $Q =[\frac{N}{2},\frac{N}{2}]$. The measurements were produced via the CPU-only mode of the implementation presented in Chapter 6. At the top of the chart, labels indicate the number of retained block states (m=4096).

The infinite-lattice algorithm starts with the four site configuration, where each block contains a single spin. In each iteration step, both blocks are enlarged by a single site making the complete system increase by two until the desired system size, N, is reached. In each iteration of the DMRG algorithm, the lowest-lying eigenvector of the

DOI:10.15774/PPKE.ITK.2014.001

corresponding superblock Hamiltonian (H_{SB}) is obtained by the iterative Davidson or Lanczos algorithms. In the presented implementation the Davidson algorithm have been considered. From the lowest eigenstate the density matrix is constructed, which carry the information how to optimally truncate the basis of the enlarged block ($m \ll q^{l+1}$) in order to keep the problem size manageable [82].

The most time-consuming part of a full iteration is the step of the Davidson routine which carries out the projection operation ($X' = H_{SB}X$). Instead of constructing and storing the enormous H_{SB} matrix of size $\mathcal{O}(m^4)$ explicitly, it is computationally favorable to obtain the projected vector X' directly from the matrices of size $\mathcal{O}(m^2)$ composing H_{SB}.

The H_{SB} can be directly expressed by the operators of the original four subsystems (*l*-1-1-*r strategy)* or by the operators of two intermediate systems (*LR strategy*), so called *enlarged blocks*, which come from the contraction of each block with its neighboring site. In the current implementation only the second strategy is investigated, however, the first one is also straightforward and will be included in the near future.

There are several practical benefits of these strategies. First of all, the memory bandwidth limited matrix-vector multiplication (BLAS Level 2) is converted to matrix-matrix multiplications (BLAS Level 3) which can be efficiently accelerated. (BLAS stands for *Basic Linear Algebra Subroutines*, which is a standard interface for linear algebraic operations.) Secondly, skipping of the explicit Kronecker multiplications not only restructures the computation, but decreases the number of operations. Finally, both strategies drastically decrease the size of the matrices which take part in the operations and thus the memory footprint of the algorithm. In case of LR and *l*-1-1-*r* strategies, the largest matrix has a size of $O((mq)^2)$ and $O((m)^2)$, respectively. The second strategy is more favorable in extreme situations when the GPU memory is limited and q (internal degrees of freedom) is large (e.g. spin-F Hubbard model with large F).

DOI:10.15774/PPKE.ITK.2014.001

5.4.1 *LR strategy*

In the *LR strategy* the H_{SB} is expressed with operators $A_\alpha^{(L)}$ and $B_\alpha^{(R)}$ defined on the left ($L := l + 1$) and right ($R := r + 1$) enlarged blocks, respectively, as

$$H_{SB} = \sum_\alpha A_\alpha^{(L)} \otimes B_\alpha^{(R)} , \tag{5.9}$$

where the index α iterates over the distinct operator combinations required to construct the superblock Hamiltonian. Exploiting Kronecker multiplication properties, the projected vector X' can be computed by matrix-matrix multiplications as

$$\tilde{X}' = \sum_\alpha A_\alpha^{(L)} \tilde{X} B_\alpha^{(R)T} , \tag{5.10}$$

where vector X of size $[B^{col} A^{col}]$ is reshaped to matrix $\tilde{X}$ of size $[B^{col}, A^{col}]$ and vector X' of size $[B^{row} A^{row}]$ is reshaped to matrix $\tilde{X}'$ of size $[B^{row}, A^{row}]$.

Algorithm 7 The computation of the projected vector X' in case of *l*-1-1-*r strategy* strategy.

Require: $size(X) = [D^{col} C^{col} B^{col} A^{col}]$
1: **function** PROJECTX_L11R(A, B, C, D, X)
2: X_1=reshape(X) **as** size(X_1)=$[D^{col}, C^{col}, B^{col}, A^{col}]$
3: **for** each (i, j) **do** $X_1'(:, :, i, j)$=$DX_1(:, :, i, j)$
4: **for** each (i, j) **do** $X_1''(:, :, i, j)$=$X_1'(:, :, i, j)C^T$
5: X_2=reshape(X_1'') **as** size(X_2)=$[D^{row} C^{row}, B^{col}, A^{col}]$
6: **for** each (i) **do** $X_2'(:, :, i)$=$X_2(:, :, i)B^T$
7: X_3=reshape(X_2') **as** size(X_3)=$[D^{row} C^{row} B^{row}, A^{col}]$
8: X_3'=$X_3 A^T$
9: X'=reshape(X_3') **as** size(X')=$[D^{row} C^{row} B^{row} A^{row}]$
10: **return** X'

In a practical implementation, Equation 5.10 operates on even smaller matrices as the operators are decomposed according to quantum numbers. Instead of a sparse matrix $A^{(L)}$, several dense matrices $A_{q_i \to q_j}^{(L)}$ are stored representing how $A^{(L)}$ transforms the subspace (sector) corresponding to q_i to the one corresponding to q_j. To compute X' in case of a given $A_\alpha^{(L)}, B_\alpha^{(R)}$ operator pair, all possible $A_{\alpha,q_i \to q_j}^{(L)}$, $B_{\alpha,q_k \to q_l}^{(R)}$ transition pairs shall be submitted to Equation 5.10, and each time only the corresponding ik and jl segments of X and X' shall be used as

$$\tilde{X}_{jl}' += A_{\alpha,i \to j}^{(L)} \tilde{X}_{ik} B_{\alpha,k \to l}^{(R)T}, \tag{5.11}$$

DOI:10.15774/PPKE.ITK.2014.001

where $\tilde{X}_{ik}$ and $\tilde{X}'_{jl}$ indicate the reshaped ik and jl segment of vector X and X', respectively. Fortunately, the reshape operation has no computational cost as the data in the memory is untouched and only the row/col sizes are changing.

5.4.2 *l*-1-1-*r* *strategy*

In the *l*-1-1-*r* *strategy* the H_{SB} (see Equation 5.9) is expressed by the operators of the four subsystems:

$$H_{SB} = \sum_{\alpha} A_{\alpha}^{(l)} \otimes a_{\alpha} \otimes b_{\alpha} \otimes B_{\alpha}^{(r)} \;, \tag{5.12}$$

where the index α again iterates over the distinct operator combinations required to construct the superblock Hamiltonian.

Similarly to the LR strategy, Kronecker multiplication properties can be exploited to compute the projected vector X' efficiently with matrix-matrix operations, however, in this case a more complicated data storage and several tensor multiplications are needed to avoid unnecessary memcopy operations. Using the procedure PROJECTX_-L11R(), which computes the projected vector X' for one matrix quadruplet and is described in Algorithm 7, the H_{SB} can be calculated as

$$X' = \sum_{\alpha} \text{PROJECTX_L11R}(A_{\alpha}^{(l)}, a_{\alpha}, b_{\alpha}, B_{\alpha}^{(r)}, X) \;. \tag{5.13}$$

In the similar manner as shown in LR strategy, $A^{(l)}, a, b$ and $B^{(r)}$ operators can be decomposed according to quantum numbers and instead of large sparse matrix operations several smaller dense matrix operations shall be submitted to Algorithm 7. Furthermore, none of the reshape operations of Algorithm 7 involves practical data movement, only the size descriptor variables are changing.

5.5 Parallelism and run-time analysis

For the Heisenberg and the Hubbard models, the run-time analysis of the DMRG algorithm is shown in Figures 5.3 and 5.4, respectively. I created the measurements with the CPU-only version of the implementation presented in Chapter 6 to investigate which parts of the algorithm shall be accelerated. As the Davidson algorithm, which is summarized in Algorithm 8, is the most time-dominant part and takes more than 97%

DOI:10.15774/PPKE.ITK.2014.001

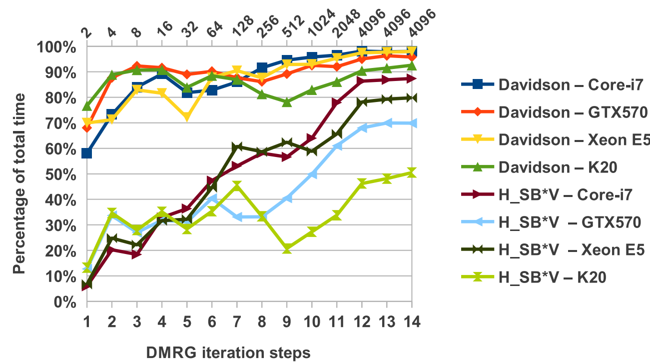

Figure 5.3: Heisenberg model: Run-time of the Davidson algorithm and its $H_{SB}V$ operation compared to the total time of a DMRG iteration step as the number of retained block states increases. CPU-only versions are indicated by Core-i7 and Xeon E5, while hybrid versions are indicated by GTX 570 and K20. At the top of the chart, labels indicate the number of retained block states (m=4096).

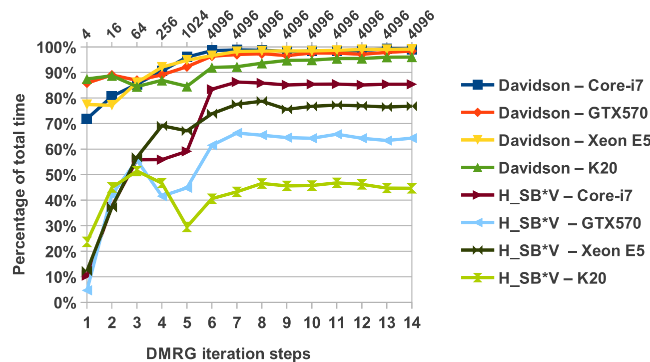

Figure 5.4: Similar to Figure 5.3 but for the Hubbard model.

of the total time, it has been chosen for acceleration. Unfortunately, the full Davidson algorithm cannot be implemented on the GPU as the problem size in real world simulations usually exceeds the GPU memory size. Instead, a hybrid approach shall be implemented, which can adjust the GPU workload according to the available GPU memory and the CPU-GPU performance ratio.

In the Davidson algorithm inherent parallelism can be observed at two levels. First, at low level, all the matrix and vector operations can be accelerated. Secondly, at the level of projection computation (line 2 in Algorithm 8), which is the most time-

DOI:10.15774/PPKE.ITK.2014.001

Algorithm 8 One iteration of the Davidson algorithm

Require: Previous $(i-1)$ basis vectors already computed.
1: **function** DAVIDSONITER(i)
2: $W(:,i) = H_{SB} \cdot V(:,i)$ ▷ BLAS-3: dgemm()
3: $B(:,i) = V^T \cdot W(:,i)$ ▷ BLAS-2: dgemv_trans()
4: $[\lambda, y] \longleftarrow$ get smallest eigvalue and vector of B
5: $x = V \cdot y$ ▷ BLAS-2: dgemv()
6: $r = -\lambda \cdot x + W \cdot y$
7: **if** norm$(r) \approx 0$ **then**
8: return with x and success
9: **else**
10: correct r
11: // orthonormalize r against V:
12: $s = V^T \cdot r$ ▷ BLAS-2: dgemv_trans()
13: $r = r - V \cdot s$ ▷ BLAS-2: dgemv()
14: normalize r and append to V
15: return without success

dominant part of the Davidson algorithm itself taking more than 75% of the total time, the projection can be computed as a sum of independent $(AX)B^T$ operations (see Equation 5.10).

At low level, the CPU part of the algorithm (regardless the GPU is enabled or not) uses the Basic Linear Algebra Subroutine (BLAS) interface and the Intel MKL Library for algebraic operations including operator contractions, inner operations of both Davidson and full diagonalization algorithms, and operator transformations. Unfortunately, in the Davidson algorithm, all the operations except the projection are BLAS level 2 matrix-vector multiplications, which are bandwidth limited and not ideal for acceleration. There is a block extension [85] of the algorithm, the so called Davidson-Liu, to determine a few of the lowest eigenvalues, where more than one candidate vectors are added at once resulting in BLAS level 3 operations, however, in the current DMRG implementation only the lowest eigenvalue is investigated. The remaining option is to store as much data in GPU memory as possible and execute the corresponding operations on GPU.

At the level of projection operation, the independence of matrix multiplications provides a straightforward hybrid parallelization and a future multi-GPU modification

DOI:10.15774/PPKE.ITK.2014.001

of the current implementation. Acceleration can be improved by developing an appropriate scheduling of the matrix operations for different matrix sizes and architectures.

5.6 Limits of the FPGA implementation

To estimate the performance of an FPGA implementation of the DMRG method, the acceleration of the projection operation expressed as a series of dense matrix multiplications (see Equation 5.11) shall be investigated. The floating-point matrix-matrix multiplication was already implemented by Kumar et al. [86] on FPGA very efficiently using the rank-1 update scheme. They demonstrated that the performance is not limited by the PCIe bandwidth, which connects the FPGA to the host CPU, and nearly full utilization of the processing elements can be reached.

The idea behind the rank-1 update approach is that instead of inner products between the rows of the left matrix and the columns of the right matrix, outer products between the columns of the left matrix and the rows of the right matrix are carried out, and resulting matrices are summarized. The advantage of the approach is that, instead of multiply–accumulate operations (MACCs), multiply-add operations (MADDs) are used, which are independent from one another and can be pipelined to reach high processing element utilization.

Using the rank-1 update approach, the processing elements can be fully utilized and the following best-case estimations can be made according to the area requirements of the floating-point units. Assuming a Virtex-7 XC7VX1140T FPGA (see Table 5.1 for details), which is one of the largest FPGA of the newest Xilinx family, approximately 193 multiply-add units can be implemented. The estimated clock frequency of multipliers is 443.65 MHz, which results in 171.2 GFLOPS computing performance. This performance achievement is similar to the performance of the mid-range GTX 570 GPU used in Section 5.4. As the development time (e.g. create the design, write the source code) in case of FPGA is still much longer than in case of GPU and the

Table 5.1: Virtex-7 XC7VX1140T FPGA feature summary

Configurable Logic Blocks		DSP	Block RAM	GTH	Max
Slices	Max Distributed RAM (Kb)	Slices	Blocks (36Kb)	Transceivers	User I/O
178,000	17,700	3,360	1,880	96	1100

DOI:10.15774/PPKE.ITK.2014.001

high-end GPUs can significantly outperform the FPGA in this problem class, the GPU architecture is the better candidate for the acceleration.

DOI:10.15774/PPKE.ITK.2014.001

Chapter 6

Hybrid GPU-CPU acceleration of the DMRG algorithm

After investigating the performance capabilities of both FPGA and GPU architectures, I chose the GPU architecture for acceleration and designed a hybrid GPU-CPU acceleration of the DMRG algorithm, from which the scientific results described in my second thesis group originate. As discussed in Section 5.5, parallelism can be exploited at various levels in the algorithm. In Section 6.1, a low level GPU acceleration of some asymmetric matrix-vector operations of the Davidson algorithm is presented describing the scientific work behind Thesis II.2. In Section 6.2, scheduling strategies are investigated to accelerate the matrix-matrix operations of the projection operation with GPU composing the scientific work related to Thesis II.1. Finally, the results of the hybrid acceleration are discussed in Section 6.3.

6.1 Accelerating matrix-vector multiplications

Jacobi-Davidson version [87] of the original Davidson algorithm [88] is a preconditioned subspace iteration technique (for more details see textbook [84]) aimed at computing a few of the extreme eigenpairs of large sparse symmetric matrices and commonly used in the DMRG implementations [66, 67]. In the presented work, the [89] version of the algorithm (available in Netlib [90]) is implemented.

In each iteration, the subspace is extended with a new basis vector ($V(:,i)$), which is stored in the memory accompanied by its projection ($W(:,i)$). As the size of these vectors can be very large (see Figures 5.1, 5.2) depending on the model and the number

DOI:10.15774/PPKE.ITK.2014.001

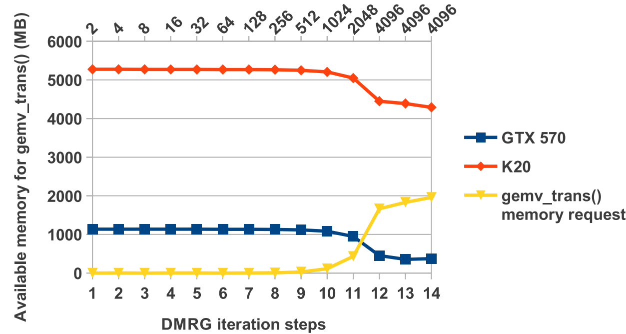

Figure 6.1: Available free GPU memory after the projection operation and the memory request of gemv_trans() in case of the Heisenberg model. As the projection operation is the most time-consuming step, its acceleration has a priority. To avoid unnecessary I/O communication, most of the matrices required by the projection are held in the GPU memory during the Davidson iterations. The acceleration of the matrix-vector operations of the Davidson algorithm can only use the remaining memory. At the top of the chart, labels indicate the number of retained block states (m=4096).

of retained block states (m), they cannot be fully stored in the GPU memory. However, in order to accelerate a matrix-vector multiplication with GPU, at least the matrix shall be stored in the GPU memory. In the current implementation the matrix of the basis vectors (V), which is used four times (see comments in Algorithm 8 on page 92) in BLAS level 2 operations, has been selected to be stored, although the storage of the projected vector matrix (W) can be added later as well.

In the implemented version of the Davidson algorithm, the number of basis vectors can be limited by a user-defined threshold to keep the memory requirements manageable. When the threshold is reached, the iteration is restarted and the previous estimate vector (see line 5 in Algorithm 8) is used as the first basis vector. In our measurements the threshold was set to 20, hence the width of matrix V was varying between 1 and 20.

In each iteration, the new basis vector is loaded to the GPU memory in the background (if there is available space) and the workload of the BLAS level 2 operations is shared between the CPU and GPU: CPU process the new basis vector, while GPU operates on the older ones. With this technique the power of both CPU and GPU can be exploited and the transfer time of the matrix can be hidden. The implementation

DOI:10.15774/PPKE.ITK.2014.001

is flexible: if there is no more space on the GPU or the CPU performance justifies it, more than one basis vector can be processed on the CPU leaving less work for GPU. As shown in Figure 6.1, where the storage requirement of V is compared with the free space available after the projection operation, V cannot be fully stored on a GTX 570 GPU even in case of the simple Heisenberg model ($m = 4096$).

There are two types of BLAS level 2 operations: $V^T X$ and VX indicated by *gemv_trans()* and *gemv()* in Algorithm 8, respectively. In the first case, the multiplier vector can be loaded in, while, in the second case, the result can be written out in smaller parts ($\sim 5e5$) to overlap with the computation.

6.1.1 Architectural motivations

To design an efficient acceleration of the gemv_trans() operation, the memory throughput and the occupancy of the streaming processors had to be considered.

As matrix-vector operations are memory bandwidth limited, the primary goal was to efficiently use the available memory bandwidth. First, to approximate the maximal memory throughput given in the GPU data sheets, memory *coalescing* had to be utilized, which means that simultaneously running threads have to read/write continuous regions of the global memory. Second, to minimize the communication overhead of the operation, the reloading of vector elements had to be minimized. If a sufficiently large cache can be allocated in the shared memory, vector elements can be reused while the different rows of the matrix are being processed. In an ideal case each vector element has to be loaded only once, however, in this case the shared memory requirement can be a critical if the number of the rows of the matrix is large.

The occupancy of the streaming processors depends on the (shared and register) memory requirement of the thread blocks. If the requirements are relatively high compared to the available resources, less thread blocks can be assigned to a streaming processor. If less thread blocks are assigned to a streaming processor, less threads can be scheduled to hide I/O latency and keep the cores busy. In the investigated cases, if the occupancy of the streaming processors decreases below a limit, the computing performance becomes the bottleneck of the overall performance instead of the memory throughput. In such cases the memory requirement of the thread blocks should be decreased to increase the occupancy.

DOI:10.15774/PPKE.ITK.2014.001

6.1.2 gemv_trans()

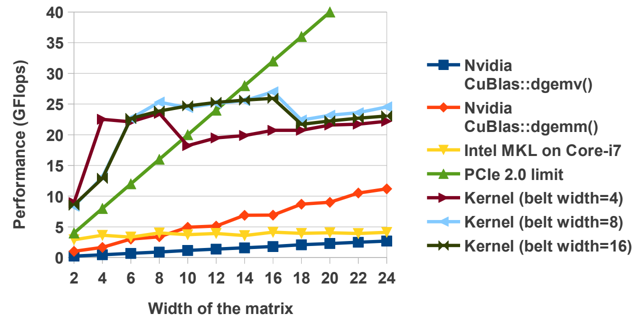

Figure 6.2: GTX 570: Performance of the presented *gemv_trans()* kernel with different belt widths is compared to the performance of the available implementations in case of matrix height $5e5$. As the belt width determines both the shared memory requirement of a thread block and the number of the reloads of the vector elements, the parameter can be used to balance the occupancy of the streaming processors and the communication overhead to reach high-performance. Additionally, the PCIe limit is displayed as PCIe throughput limits the performance of the GPU acceleration if the transfer of the multiplier vector cannot be avoided. Note that in the current design only the matrix elements have to be present in the GPU memory, the position of the vector elements is optional. If the application requires, they can be loaded overlapped with the computations.

In case of gemv_trans(), both MKL and CuBlas libraries give poor performance for the special asymmetric matrix size ($\sim 5e5$x20) required in our application (see Figures 6.2, 6.3, 6.4, 6.5 and 6.6), therefore, I designed a new CUDA kernel for the operation. The presented results are measured without data communication, as in case of line 3 the multiplier vector is already in the GPU memory providing ideal acceleration. To estimate the performance of line 12, where the multiplier vector has to be loaded, the limiting factor of the PCIe 2.0 is also displayed. In this case the overall performance cannot exceed the PCIe limit even with overlapped communication. In both cases, the estimation of the overall acceleration shall be carried out in an integral fashion as in each iteration the thickness of the matrix is increased by one until a user defined limit (20 in the presented DMRG test-cases) is reached.

The basic idea of the new kernel (see Algorithm 9) can be summarized as follows. Each thread is associated with a column of the matrix. Each thread loads the

DOI:10.15774/PPKE.ITK.2014.001

corresponding vector element and multiplies the elements of the associated column. As threads of a warp load consecutive elements of the vector and the matrix, the coalesced reading is obvious. If the number of threads (grid size * thread block size) is less than the length of the matrix, each thread is associated with a new unprocessed column (coalesced readings again) as long as there is any. After processing a new column each thread accumulates the results to the results of the first column. Finally, the accumulated results shall be summed across the threads, which can be efficiently done via a sum reduction [91] in shared memory. If the belt is smaller than the width of the matrix, the whole procedure can be repeated (outer loop).

Algorithm 9 Pseudocode of the proposed kernel for asymmetric gemv_trans()

Require: *thread_number* is the global index of a thread
1: **function** GEMV_TRANS(MTX, MTX_WIDTH,
2: MTX_LEN, VEC)
3: Allocate shared memory for a thread block(*s_block*)
4: **for** each *belt* **do**
5: Fill *s_block* with zeros
6: **for** (i = thread_number; i < mtx_len;
7: i += num_of_threads) **do**
8: Load vec[i] to a private memory (*p_vec*)
9: **for** each element of ith column of *belt* **do**
10: Load element, multiply with *p_vec* and accumulate the product to *s_block*
11: Sum reduction in *s_block*
12: **if** this is the first thread of the block **then**
13: Save first column of *s_block*

The size of the shared memory requirement of a thread block, which is equal to the size of a thread block multiplied with the width of the belt, can be a limiting factor of the performance because in case of large shared memory usage less thread blocks can be assigned to one physical multiprocessor. In the presented measurements the optimal width of the belt has been investigated, however, even with the optimal width the performance decreases as the width of the matrix increases. For extreme, asymmetric matrices, which are used in our application, significant speed-up (x4-5) can be reached compared to the CuBlas library, however, as the matrix tends to be more symmetric the

DOI:10.15774/PPKE.ITK.2014.001

performance of the CuBlas::dgemm() operation (red line in Figures 6.2-6.6) increases and exceeds the performance of the new kernel.

In case of the new Kepler architecture (K20), in which Streaming Multiprocessor has significantly more CUDA Cores than the SM of Fermi GPUs (GTX 570), the per-multiprocessor warp occupancy shall be increased to use all the available cores [92]. A new warp-level intrinsic called the shuffle operation can be used to decrease the shared memory requirement of the sum reduction algorithm to increase the occupancy. If shuffle operation is enabled, each thread accumulates the partial products (line 10 in Algorithm 9) in a private memory, and the shuffle operation is used to summarize the results of the threads of the same warp. Consequently, only one column per warp has to be stored in the shared memory, which decreases the shared memory requirement to the number of warps in a block multiplied with the width of the belt. In practice, the cycle at line 9 is unrolled with macros as allocation of static arrays in private memory is not possible. In Figures 6.4, 6.5 and 6.6 the results of the new kernel extended with the shuffle operation are displayed. The width of the optimal belt is slightly increased as the shared memory request is decreased. Unfortunately, the shuffle operation provides only a small performance gain in case of our kernel (compare Figures 6.3 and 6.5).

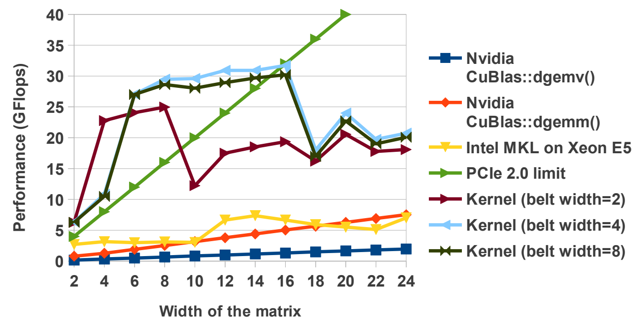

Figure 6.3: K20, no shuffle operation in the kernel: Performance of the presented *gemv_trans()* kernel with different belt widths is compared to the performance of the available implementations in case of matrix height $5e5$. As the belt width determines both the shared memory requirement of a thread block and the number of the reloads of the vector elements, the parameter can be used to balance the occupancy of the streaming processors and the communication overhead to reach high-performance. Additionally, the PCIe limit is displayed as PCIe throughput limits the performance of the GPU acceleration if the transfer of the multiplier vector cannot be avoided.

DOI:10.15774/PPKE.ITK.2014.001

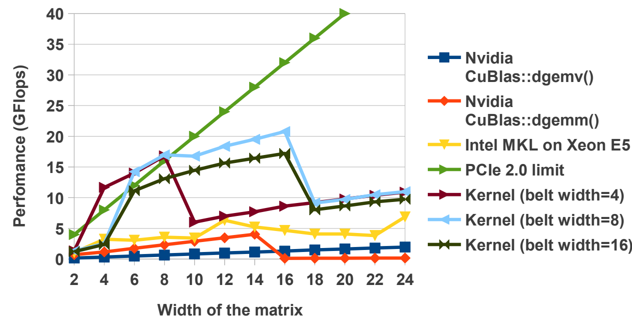

Figure 6.4: K20, shuffle operation enabled: Performance of the presented *gemv_trans()* kernel with different belt widths is compared to the performance of the available implementations in case of matrix height $1e5$. As the belt width determines both the shared memory requirement of a thread block and the number of the reloads of the vector elements, the parameter can be used to balance the occupancy of the streaming processors and the communication overhead to reach high-performance. Additionally, the PCIe limit is displayed as PCIe throughput limits the performance of the GPU acceleration if the transfer of the multiplier vector cannot be avoided.

With the experiments presented in Figures 6.2-6.6, the optimal belt width was investigated in the function of the matrix width in case of different problem sizes and GPU version. As the belt width determines both the shared memory requirement of a thread block and the number of the reloads of the vector elements, it should be set to balance the occupancy of the streaming processors and the communication overhead. The performance of the kernel is also affected by the width of the input matrix, which is a consequence of a certain limitation of the CUDA environment. As kernels cannot allocate static arrays in the private memory, the cycle at the 9th line of Algorithm 9 is unrolled with macros. Each macro has to check whether there is enough row in the matrix to proceed with the computations. Each extra checking creates an overhead, which becomes significant, when several matrix rows are missing, that is, when the remainder after the division of the matrix width by the belt width is large.

The performance of the kernel is mainly dominated by the speed of the coalesced reading of the matrix elements. The gemv_trans() operation is memory bandwidth limited in both CPU and GPU architectures, but it is better to compute it on GPU because the bandwidth on GPU (e.g. GDDR5 in GTX 570: $152GB/s$ or GDDR5 in K20: $208GB/s$) is usually higher than on CPU (e.g. DDR3-1333 in dual channel

DOI:10.15774/PPKE.ITK.2014.001

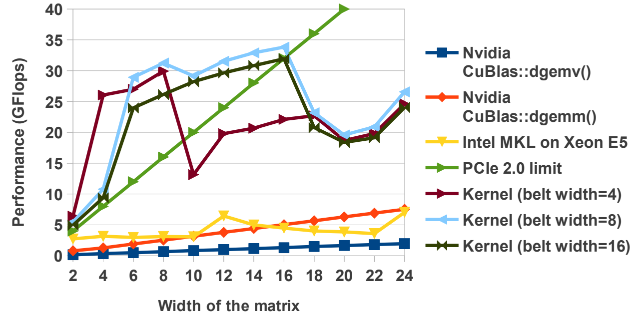

Figure 6.5: Similar to Figure 6.4 but for matrix height $5e5$.

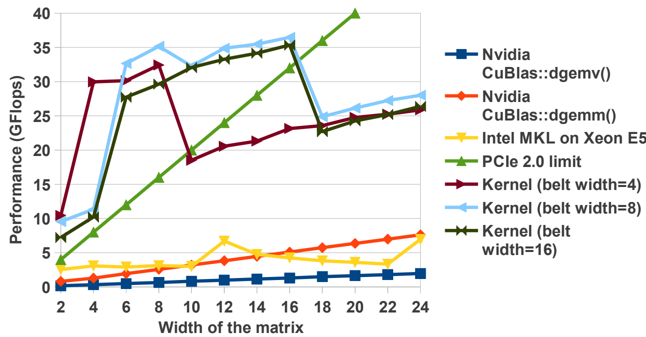

Figure 6.6: Similar to Figures 6.4 and 6.5 but for matrix height $1e6$.

with Core-i7: $21.2GB/s$ or DDR3-1066 in quad channel with Xeon E5: $34.1GB/s$). The maximal memory throughput reached by the new kernel (measured with matrix size $16 \times 5e5$) was $114.7GB/s$, $134.8GB/s$, and $143.7GB/s$ on GTX 570, on K20 without shuffle, and on K20 with shuffle, respectively.

In the presented DMRG implementation, the shuffle operation is enabled and the kernel with the best belt width is invoked based on the matrix width. The results of the acceleration of the selected BLAS level 2 operations of the Davidson algorithm on the Xeon E5 + K20 architecture are summarized in Table 6.1. (On the GTX 570 card the memory is too small to accelerate other operations besides the projection.) Line 3 is accelerated well as no extra communication is needed while the other operations are either limited by the PCIe or the DDR3 throughput.

DOI:10.15774/PPKE.ITK.2014.001

6.1.3 gemv()

The gemv() operation can be efficiently accelerated by the standard CuBlas library even in case of asymmetric matrices (see Figure 6.7). Unfortunately, merging of CPU and GPU results is slow on one CPU thread and is the bottleneck of the acceleration. The implementation can be improved by multithreaded merging to enable quad channel memory or by computing everything on the GPU, however, this is not always possible.

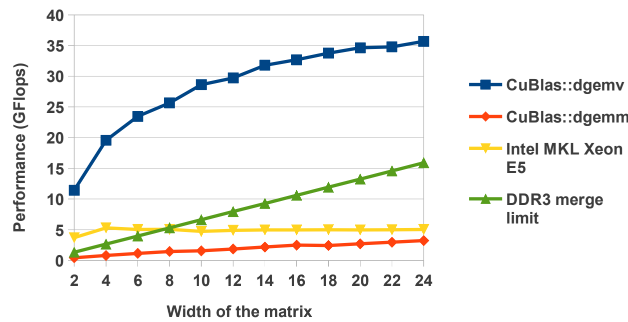

Figure 6.7: Performance of the gemv_normal() operation of the available implementations in case of K20. Additionally, the DDR3 limit is displayed, because the merge operation is limited by the DDR3 throughput in hybrid mode. The gemv() function of the CuBLAS library provides acceptable acceleration compared to the CPU, however, the merging of the results of the two architecture limits the final performance.

6.2 Accelerating projection operation

The acceleration of the independent $(AX)B^T$ operations is based on the observation that A and B matrices are already available before the Davidson algorithm starts and do not change during the Davidson iterations. The necessary $(AX)B^T$ operations are described by a list of *operation records*, in which each record contains all the necessary information to compute an operation like Equation 5.11. For example, it stores the information from which segment of X (*input*) to which segment of X' (*output*) the operation transforms.

The host side algorithm to handle the operation records is summarized in Algorithm 10. During the construction of the operation records, the workload associated

DOI:10.15774/PPKE.ITK.2014.001

Table 6.1: Runtime of the accelerated matrix-vector operations of the Davidson algorithm in seconds (see Algorithm 8 on page 92 for the line numbers). Lines 3 and 12 correspond to the gemv_trans() operation. In case of line 3, the vector elements are already in the GPU memory, therefore, the PCIe bandwidth is not a limiting factor. In case of lines 5 and 13, the poor acceleration is due to the merging of results, which is limited by the DDR3 bandwidth.

		Xeon E5	Xeon E5 + K20	speedup
Heisenberg model	Line 3	20.21	4.64	4.36
	Line 5	19.07	10.45	1.83
	Line 12	20.05	5.94	3.38
	Line 13	17.55	9.68	1.81
Hubbard model	Line 3	113.80	22.66	5.02
	Line 5	94.29	54.22	1.74
	Line 12	114.00	37.67	3.03
	Line 13	87.29	50.21	1.74

to each output is computed. (Multiple operations can use the same input or write the same output segment.) Next, the operation records are partitioned between CPU and GPU based on the performance ratio of the two architectures. To avoid merging of outputs, all operation records corresponding to the same output shall be computed on the same architecture, however, to create a balanced workload partitioning, this is not always possible. During partitioning, the output associated to the largest workload is selected for GPU iteratively as long as the desired workload ratio is not exceeded. If the reached workload ratio is far from the desired, the operation records of the output associated to the next largest workload are partitioned between the two architectures.

Algorithm 10 Host side algorithm to handle the operation records

1: Create operation records and determine the workload (FLOP) for each output.
2: Partition the operation records between CPU and GPU based on their performance ratio and the output workload statistics.
3: Selects scheduling strategy for the operations to be computed on GPU.
4: Set-up the workload for GPU based on the selected strategy.

After partitioning, the proper scheduling strategy is selected based on the memory requirements of the operation records. Three different strategies can be selected for three different uses cases, however, currently only the first two, more complex strategies have been implemented and tested. The first strategy (*4Streams*) is designed for small problem size, when all A, B, X, X' matrices and temporary matrices T for

DOI:10.15774/PPKE.ITK.2014.001

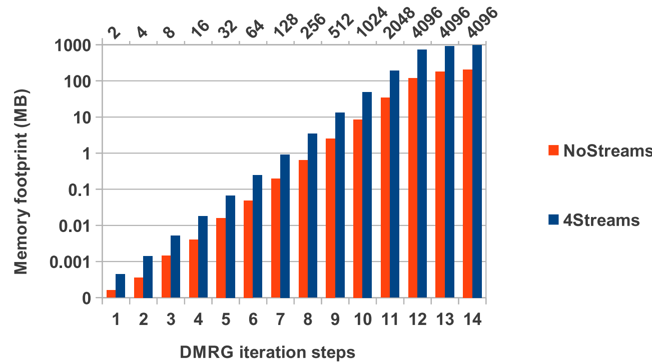

Figure 6.8: GPU memory footprints of the two strategies are compared in case of the Heisenberg model. At the top of the chart, labels indicate the number of retained block states (m=4096).

storing intermediate results can be held in the GPU memory. The second strategy (*NoStreams*) is designed for medium-sized problems, where all A, B and X' matrices can be stored in GPU memory, but from X and T only the processed matrices are allocated. In case of extra-sized problems, a third strategy (*NoStreamAndStorage*) can be designed, in which even A and B matrices cannot be fully stored in the GPU memory. The memory footprint of the matrices in case of different strategies are shown in Figure 6.8. In the demonstrated examples the first strategy was available for all the DMRG iterations as the GPU cards had enough memory.

6.2.1 Architectural motivations

To design efficient strategies for the different use cases of the $(AX)B^T$ operation, the limitations of the CUDA kernel scheduling and the bandwidth of the PCI express interface have to be considered. First, as smaller matrices do not provide enough computation to keep all the GPU cores busy, more $(AX)B^T$ operations have to be executed simultaneously to maintain high performance. Second, the operation records shall be scheduled to hide the transfer time of matrices communicated through the PCI express interface.

The $(AX)B^T$ operations are implemented using the CuBLAS libary [93], which is a BLAS implementation dedicated for Nvidia GPUs. In the demonstrated strategies, two important features of the GPUs are exploited, which are provided via the CUDA

DOI:10.15774/PPKE.ITK.2014.001

driver [94] and also accessible through the CuBLAS library. The first feature is that multiple CUDA kernels can be executed simultaneously on the GPU, while the second feature is that memory I/O operations can be executed in the background. From the aspect of programming, both features can be accessed via the CUDA streams. Streams are sequences of operations that execute in issue-order, but operations in different streams may run concurrently or interleaved.

Although the CUDA environment allow the parallel execution of kernels, it does not guarantee the parallel execution unless the kernels are scheduled and issued in a proper order by the programmer. To maintain the simultaneous execution of multiple kernels, an extra scheduling of the operation records has to be done, which pays attention to the limited scheduling capabilities of the CUDA kernel dispatching mechanism described in case of the 4Streams strategy.

6.2.2 Scheduling strategies

In the 4Streams strategy, there is enough GPU memory to execute several $(AX)B^T$ simultaneously. One stream is created for each output and operations corresponding to a given output are assigned to the same stream to avoid interference. For each stream, a sufficiently large temporary matrix is allocated to store the temporary result of AX.

CUDA operations are dispatched to *hardware queues* in issue order [94]. To enable asynchronous concurrent kernel execution in CUDA environment, memory transfers and kernels shall be issued in a breadth-first order. Inside the engine (kernel) queue an operation is dispatched if all preceding calls in the same stream have been completed and all preceding calls of the same queue have been dispatched. Consequently, to avoid blocking calls, kernels of the same streams shall not be issued immediately after each other. As one $(AX)B^T$ operation consists of two kernels, the kernel calls shall be separated and interleaved with kernels of operations of other streams. To reach four parallel streams (hence the name of the strategy), kernels from four different streams shall be interleaved. That is, if we have four operation records associated to different streams, we have to issue the first kernel of each operation before we continue with the second kernels.

Overlapping of the transfer time of input segments with kernel execution makes further constraints on the order of the operation records: only those operation records

DOI:10.15774/PPKE.ITK.2014.001

Algorithm 11 Grouping operation records in case of 4Streams strategy

```
 1: function ORDERANDGROUPRECORDS(records, maxstream)
 2:     Sort records by input frequency.
 3:     setVisitedRecords.clear()
 4:     for each record i do
 5:         if i.stream ∈ setVisitedRecords then
 6:             for each record j following i do
 7:                 if j.stream ∉ in setVisitedRecords then
 8:                     swap(i,j) and break
 9:         if i.stream is ∉ in setVisitedRecords then
10:             vecGroup.last().insert(i)
11:             setVisitedRecords.insert(i.stream)
12:             if setVisitedRecords.size()=maxstream then
13:                 vecGroup.add(new Group)
14:                 setVisitedRecords.clear()
15:         else
16:             vecGroup.add(new Group)
17:             vecGroup.last().insert(i)
18:             setVisitedRecords.clear()
19:             setVisitedRecords.insert(i)
    return vecGroups
```

shall be issued which use already loaded input segments. To be able to interleave different streams, it is favorable to load the input segment first which is used by the most streams.

Algorithm 12 Dispatching operation records

1: **for** each group g **do**
2:     **for** each record i in g **do**
3:         Init copy of input segment X_i (when first used)
4:         Init $T_i = (A_i X_i)$
5:     **for** each record i in g **do**
6:         Init $X_i' = T_i B_i^T$
7: **for** each output segments of X' **do** Init copy back.

In case of 4Streams strategy the reordered operation records are grouped (see Algorithm 11) such a way that the issuing of the kernels belonging to the same group can be interleaved (see Algorithm 12). First, operation records are sorted to load the more

DOI:10.15774/PPKE.ITK.2014.001

frequently used input segments earlier. Then, records are iterated and each record is potentially swapped backward to create groups of four consecutive operations belonging to four different streams. In practice, some technical constraints have been added to slightly alter the swapping behavior, however, it is not discussed here for the sake of simplicity.

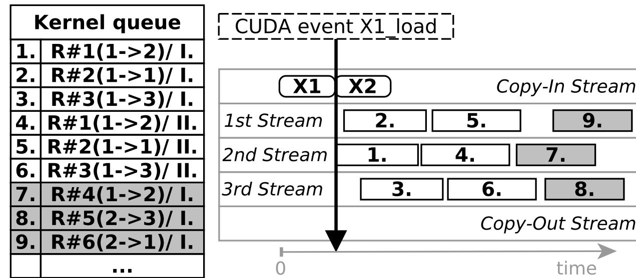

Figure 6.9: Interleaved operation records (on the left) and the resulting parallel execution (on the right). Each record contains two kernel calls (see line 4 and 6 in Algorithm 12), which are indicated by roman letters. The lth kernel of the ith record is named as $R\#i(j \rightarrow k)/l$, where j^{th} and k^{th} indicate the affected input and output segments, respectively. The kernel queue illustrates the issue order of the kernels. The kernels of the first group are colored by white, while grey color indicates some of the kernels of the second group. A CUDA event is also displayed to demonstrate that the first kernel does not start until the first segment is loaded. Note that the 9th kernel cannot start until all the previously issued kernels have been started.

CUDA operations are launched according to the operation records as summarized in Algorithm 12. For the sake of brevity, the synchronization between the streams is not shown as it can be implemented with CUDA events in a straightforward way. (For example, if each operation waits for the transfer event specific to the utilized input segment and the operations are ordered properly, the transfer and computation time can be overlapped.) The same code can be used for both strategies as the NoStreams strategy can be represented by groups which contain only one operation record. In case of NoStreams strategy, the preparation of the operation records is much simpler and contains only the sorting by input frequency to reach I/O overlap with computation. To illustrate the interleaved kernel calls, the overlapped I/O communication and the parallel kernel execution, a schematic diagram of a simplified example is shown in Figure 6.9.

DOI:10.15774/PPKE.ITK.2014.001

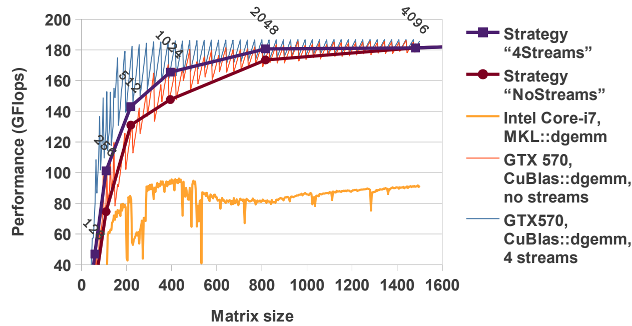

Figure 6.10: GTX 570, Heisenberg model: Performance of the two strategies is compared. Additionally, the performance of CuBLAS and MKL dgemm() in reference measurements is displayed as the function of matrix size. Labels indicate the number of retained block states at the displayed DMRG iterations.

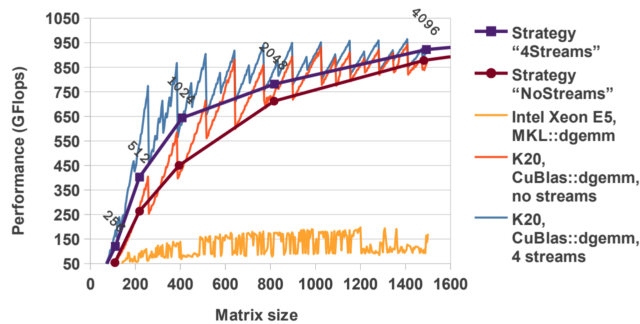

Figure 6.11: Similar to Figure 6.10 but on K20 architecture.

The performance of the two strategies is compared in Figures 6.10 and 6.11. Significant improvement can only measured at medium sized matrices (100-800 for GTX 570 and 100-1500 for K20), in which case several operations shall be executed concurrently to keep all the CUDA cores busy. Slightly bigger gain can be observed in case of K20 GPU, which has 2496 Kepler CUDA cores as opposed to GTX 570 having only 480 Fermi CUDA cores. Operations on large matrices ($\sim$$1500$ for GTX 570 and $\sim$$3000$ for K20) provide enough work for each CUDA core to approach the theoretical maximum double performance (180 GFlops for GTX570 and 1.17 TFlops for K20) without streams.

DOI:10.15774/PPKE.ITK.2014.001

Table 6.2: Total time of strategies is compared. Although, the 4Streams strategy produced significant acceleration in case of smaller ($400 \sim 600$) matrices, the total run-time of the algorithm was not decreased significantly because the average matrix size was relatively large in the investigated models. Two important tendencies can be observed even in the presented run-times. First, on both GPUs larger acceleration was reached in case of the Heisenberg model, which was due to the fact that the basis size scaled with a smaller exponent compared to the other model. Second, in case of K20, which has significantly more computing cores, both model profited more from the 4Streams strategy.

	Model	NoStream (sec)	4Streams (sec)	decrease
GTX570	Heisenberg	671.54	652.58	2.82%
	Hubbard	2980.27	2957.82	0.75%
K20	Heisenberg	244.67	227.33	7.09%
	Hubbard	1056.33	1012.56	4.14%

The two strategies are also compared by the run-time of the simulated models in Table 6.2. In case of K20, the concurrent kernel execution has a slightly greater benefit, however, in both models, operations on larger matrices, where concurrency has no benefit, dominates the run-time. In models where more symmetries are enabled, the size of the matrices tends to be smaller, consequently, in these models the concurrency also tends to be more effective.

The performance results of the full projection computation including both CPU and GPU computations are shown in Figures 6.12, 6.13, 6.14 and 6.15. The quality of the acceleration is highly affected by the applied workload ratio, which depends on the performance ratio of CPU and GPU at the given matrix size. In the configuration file different ratios can be set for different matrix sizes and in each DMRG iteration the user-defined ratio is selected according to the average matrix size of the operation records. If the workload is properly distributed 257.8 GFlops ($\times 3.2$ speed-up) and 1071.1 GFlops ($\times 6.1$ speed-up) can be reached on GTX 570 and on K20, respectively.

6.3 Implementation results

The DMRG algorithm was implemented in C/C++ and can be complied in a CPU-only and a hybrid CPU-GPU mode. In the CPU-only mode, all the basic linear algebra subroutines (BLAS) are accelerated with the Intel MKL [83] library, while in

DOI:10.15774/PPKE.ITK.2014.001

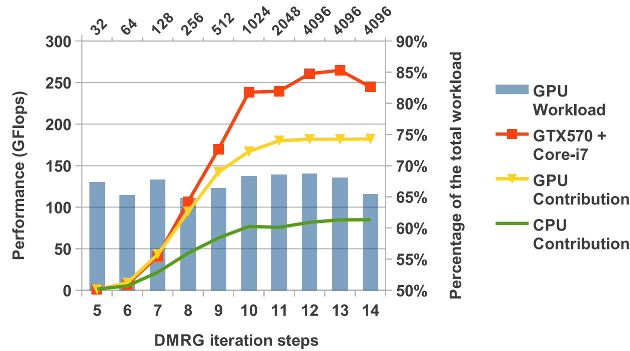

Figure 6.12: GTX 570, Heisenberg model: Performance results of the hybrid CPU-GPU acceleration of the projection operation. Blue bars associated to the secondary vertical axis indicate the ratio of the current GPU workload. At the top of the chart, labels indicate the number of retained block states (m=4096).

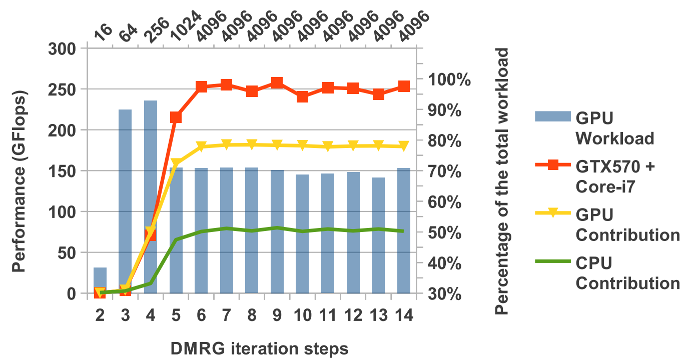

Figure 6.13: Similar to Figure 6.12 but for the Hubbard model on GTX 570.

the hybrid mode some of the operations are executed on GPU using the CUDA 5.0 environment [94]. Matrix-matrix multiplications related to the projection operation of the Davidson algorithm are executed via the NVidia CuBlas [93] library, while some asymmetric matrix-vector multiplications are executed via the proposed CUDA kernels.

The implementation has been tested both on a mid-range (Intel Core-i7 2600 3.4 GHz CPU + NVidia GTX 570 GPU) and on a high-end configuration (Intel Xeon E5-2640 2.5 GHz CPU + NVidia K20 GPU); the results are displayed in Table 6.3 and 6.4, respectively. All CPU-only measurements have been executed with multithreading

DOI:10.15774/PPKE.ITK.2014.001

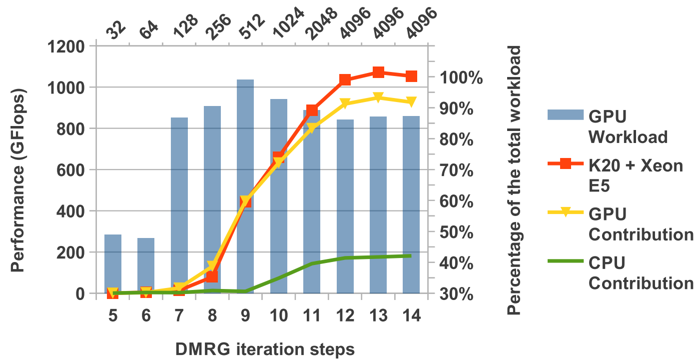

Figure 6.14: Similar to Figures 6.12 and 6.13 but for the Heisenberg model on K20.

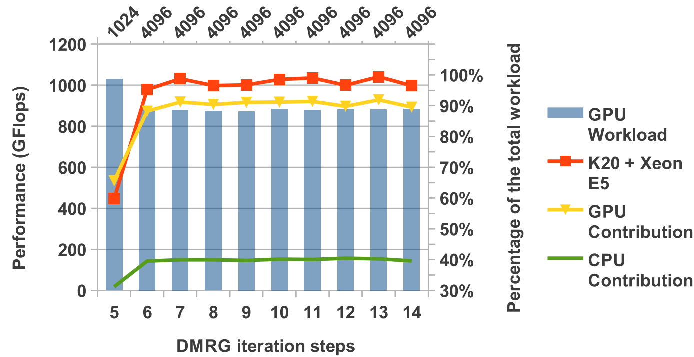

Figure 6.15: Similar to Figures 6.12, 6.13 and 6.14 but for the Hubbard model on K20.

enabled (4 threads on Core-i7 and 6 threads on Xeon E5). The mid-range configuration with GPU is approximately 2.3-2.4 times faster than without GPU, while the high-end configuration is accelerated by 3.4-3.5 times using the GPU. A change from a mid-range, multithreaded CPU to a high-end CPU+GPU configuration can produce 6.5-7 times acceleration. The main parameters of the utilized GPU cards are summarized in Table 2.2 on page 16.

To support the comparison of the results of the two investigated models, the key parameters affecting computational complexity are summarized in Table 6.5. Using the same number of retained block states, the Hubbard model has larger values for all key parameters except the maximum sector size and the maximum matrix size. In case of the Hubbard model, more symmetries are exploited, which results in smaller sectors

DOI:10.15774/PPKE.ITK.2014.001

Table 6.3: Heisenberg model: final timings compared

	Time(sec)	Speed-up compared to	
		Core-i7	Xeon E5
Core-i7	1489.64	1	0.53
Core-i7 + GTX 570	652.58	2.28	1.21
Xeon E5	789.65	1.89	1
Xeon E5 + K20	227.33	6.55	3.47

Table 6.4: Hubbard model: final timings compared

	Time(sec)	Speed-up compared to	
		Core-i7	Xeon E5
Core-i7	7210.72	1	0.48
Core-i7 + GTX 570	2957.82	2.44	1.16
Xeon E5	3433.16	2.10	1
Xeon E5 + K20	1012.56	7.12	3.39

Table 6.5: Model comparison in case of Xeon E5 + K20.

	Heisenberg	Hubbard	ratio
Time(s)	244.67	1067.89	4.36
Flop	1.22E+014	4.89E+014	4.01
Max H_{SB} size	12.24E+06	15.32E+06	1.25
Max Sector size	4.00E+06	3.47E+06	0.87
Average number of sectors	9.36	50.71	5.42
Max matrix size	1704.23	1145.24	0.67
Peak GPU memory footprint	950.47	1155.48	1.22
Average number of Davidson iterations using random starting vector	60.79	122.43	2.01

and, consequently, smaller matrices.

In case of K20, the acceleration of the projection and the matrix-vector operations is compared in Figures 6.16 and 6.17. The projection is accelerated by 5.7 times

DOI:10.15774/PPKE.ITK.2014.001

which is in accordance with the theoretical performance capabilities of the two architectures. Currently on Xeon processor (see Figure 5.3) the projection operation is only accounted for 75% of the total run-time, therefore, the overall acceleration is also affected by the rest of the operations of the Davidson algorithm. Fortunately, as the number of retained states (m) increases the time-dominance of the projection also increases, which anticipates even better acceleration for real-world simulations with large m.

As the acceleration of the full Davidson algorithm can be limited by the GPU memory, an adaptive solution shall be implemented which accelerates as much of the algorithm as possible. Currently four matrix-vector operations of the algorithm is accelerated in case of sufficient GPU memory, however, acceleration of the rest of the operations will also be implemented later.

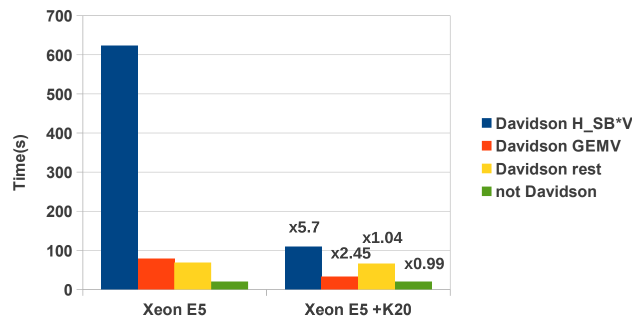

Figure 6.16: K20, Heisenberg model: Acceleration of different parts of the algorithm is compared for $m = 4096$.

6.4 Summary

In the chapter the first hybrid CPU-GPU acceleration of the DMRG algorithm was presented including the acceleration of the first and the second most time-consuming part of the algorithm, the projection operation and some of the matrix-vector multiplications of the Davidson iteration.

I proposed a new scheduling for the AXB^T operations of the projection operation, from which Thesis II.1 originates. In the scheduling two strategies can be se-

DOI:10.15774/PPKE.ITK.2014.001

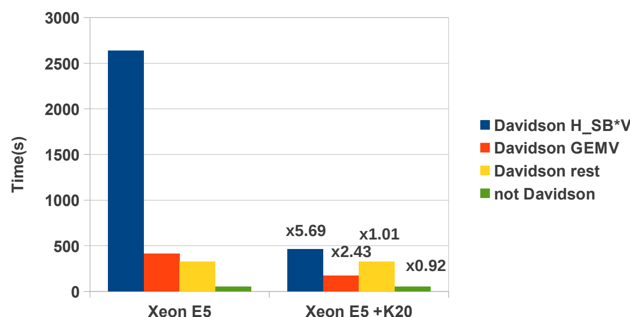

Figure 6.17: K20, Hubbard model: Acceleration of different parts of the algorithm is compared for $m = 4096$.

lected based on the average size of the matrices constructing the AXB^T operations. I designed an algorithm for each strategy to order the operations: a simple ordering to overlap computation and communication in the single-threaded strategy and a more complex ordering (see Algorithm 11) to consider also the limitations of the CUDA kernel dispatching mechanism in the multi-threaded strategy. The presented acceleration is the first GPU-based acceleration of the projection operation. The significance of the proposed solution is that the GPU can be operated with high utilization during the key operation of the DMRG algorithm. The primary application of the scheduling is the presented DMRG implementation, with witch the simulation time of certain quantum chemical systems can be significantly shortened. Further applications are possible in similar techniques, e.g. in Tensor Network (TN) methods [97], where multiplications with a Hamilton operator defined on a contracted Hilbert space have to be computed.

I designed a new algorithm for GPU to accelerate asymmetric transposed matrix-vector multiplication, which corresponds to Thesis II.2. The presented algorithm significantly outperformed the reference libraries in the extremely asymmetric case required in the Davidson iteration of the DMRG algorithm. The key feature of the algorithm is a flexible parameter allowing to find a practical balance between the communication overhead and the shared memory requirement of the kernel. The primary goal of the presented algorithm was to accelerate some of the matrix-vector operations of the Davidson iteration of the DMRG algorithm, however, the proposed acceleration can be used in any application where asymmetric matrix-vector operations have to be

DOI:10.15774/PPKE.ITK.2014.001

computed on GPU.

The next NVIDIA GPU architecture is called Maxwell. Maxwell GPUs are currently not available in the high-performance Tesla product line, but based on the GeForce products already using the Maxwell architecture, some observations can be made. The new architecture is focusing on the power efficiency of streaming multiprocessors and increases their occupancy even if less parallelism is available. Although the number of cores per streaming processor has been reduced to a power of two, the number of streaming processors has been increased and the total number of cores nearly doubled. To increase the occupancy, the shared memory in each multiprocessor has been also increased. In case of the acceleration of the projection operation, the 4Streams strategy can produce better results as the number of cores has been further increased. Furthermore, as the performance of the current Tesla cards will be doubled, the current performance advantage of the GPU compared to the CPU will remain for the next generation as well. In case of the acceleration of the memory bandwidth limited matrix-vector operations, no clear estimations can be given because the memory capabilities of the new Tesla cards are still not known. On the one hand, the advent of the DDR4 memories will double the memory bandwidth of CPUs, however, on the other hand, GPU manufactures are also searching the possibilities (e.g. using stacked DRAMs like Micron's hybrid memory cube) to improve their solutions.

DOI:10.15774/PPKE.ITK.2014.001

Chapter 7

Summary of new scientific results

7.1 New Scientific Results

The statements of the dissertation are grouped into two categories: the first group deals with the construction of locally controlled arithmetic units from synchronous data-flow graphs, while the second group is focusing on the first implementation of the DMRG algorithm on modern parallel architectures.

Thesis I I designed a local control to improve the operating frequency of the FPGA implementation of synchronous data-flow graphs and gave a method to determine the number and the topology of locally controlled components in the design space of speed and area.

I.1 I designed and implemented a distributed local control to avoid global control signals and increase the operating frequency of the control unit at the expense of a moderate area increase. [3]

I experimentally showed that global control signals of the arithmetic unit of synchronous data-flow graph based FPGA implementations (e.g. numerical solution of partial differential equations) are the bottlenecks of the operating frequency of the whole circuit if the number of I/Os of the arithmetic unit is large. I designed and implemented a locally distributed control for the arithmetic units of the aforementioned applications to avoid the blocking global signals at the expense of a moderate area increase.

DOI:10.15774/PPKE.ITK.2014.001

To investigate the trade-off between speed and the number of I/Os, I measured the operating frequency of the proposed control logic without floating-point units in the function of the number of I/Os. On a Virtex-6 FPGA, which is designed for high-performance computations, a control restricted to maximally 10 I/Os can reach 510 MHz frequency, approximately 20% more than a control handling 20 I/Os. Assuming 450 MHz frequency for the rest of the circuit (e.g. floating-point units), the restricted control can be operated without holding back the whole circuit. For the control, it is worth to target a slightly higher theoretical frequency than the minimal 450 MHz because operating frequencies are typically much lower in practical designs, where floating-point units are also implemented.

I designed an optimization procedure to determine the locally controlled components of the arithmetic unit by partitioning the data-flow graph, if the number of I/Os exceeds the threshold required for fast operation. The resulting partition classes can be controlled independently, however, extra synchronizing First-In-First-Out (FIFO) buffers are required between the classes, which increase the number of utilized configurable logic blocks (area requirement of the circuit). I proposed a optimization problem to minimize the number of extra FIFOs when the data-flow graph is partitioned to meet the I/O constraints required for high performance operation.

I.2 I developed a greedy partitioning algorithm, which outperformed one of the popular state-of-the-art partitioning algorithms in case of the proposed optimization problem. [4]

Regarding a computational fluid dynamics (CFD) application, I experimentally showed that partitioning objectives alone are not sufficient to reach high operating frequency, and placement objectives shall be considered as well. I designed a simple greedy algorithm which takes placement objectives into consideration and supports the manual tuning of the placement phase of high-level synthesis. Using the greedy algorithm with manual placement constraints, the design reached approximately 370 MHz operating frequency in case of a single precision CFD test case outperforming the results of the general-purpose hMetis [53] algorithm by approximately 13%. Without manual placement constraints, the same design reached 328 and 296 MHz frequency in case of single and double precision, respectively.

DOI:10.15774/PPKE.ITK.2014.001

I.3 I developed a new graph partitioning algorithm which incorporates both partitioning and placement objectives to improve operating frequency even without manual placement constraints. [1, 5–7]

I proposed a new high-level synthesis approach, which, contrary to the traditional step-by-step strategy, incorporates placement information already at the partitioning step, and designed a new partitioning algorithm implementing the approach using simulated annealing. I evaluated the algorithm in two complex CFD test cases by measuring the operating frequency of the generated circuit in the function of the maximal I/O connection of the clusters. Maximal speed-up (15-25%) compared to the unpartitioned case was reached, when the maximal number of I/Os was set to 9 or 10. Both CFD arithmetic units reached approximately 320-325 MHz in case of double precision.

Thesis II To improve the performance of the first hybrid CPU-GPU implementation of the Density Matrix Renormalization Group (DMRG) algorithm, I designed a scheduling algorithm for the matrix-matrix multiplications of the most time-consuming step, and developed a new algorithm for asymmetric matrix-vector multiplication in GPU.

I analyzed the runtime of the algorithm and found the projection operation of the iterative diagonalization method (Davidson) to be the most time-consuming step, which can be rephrased as a sequence of dense matrix-matrix multiplications. I investigated the performance of GPU and FPGA in case of matrix-matrix multiplication, and found that the operation can be implemented on both architectures with high utilization, however, assuming full utilization of both architectures, the GPU is approximately 5 times faster than the FPGA. I created a high-performance hybrid GPU-CPU acceleration of the algorithm in CUDA environment, which is the first kilo-processor implementation of the algorithm and is approximately 3.5 times faster than the high-end, CPU-only version.

DOI:10.15774/PPKE.ITK.2014.001

II.1 I designed a new scheduling algorithm for the matrix multiplications of projection operation, which is the most time-consuming part of the DMRG algorithm, to maintain high utilization of the GPU in case of different matrix sizes. [8, 2]

I investigated the size of the matrices participating in the matrix-matrix multiplications in case of the Heisenberg and the Hubbard models, which utilized different number of symmetries. I found that the size of the matrices varies widely inside and across the iterations of the algorithm, and the average matrix size is affected by the model and the number of symmetries applied.

I measured the performance of CPU and GPU in case of matrix-matrix multiplication in the function of matrix size using the MKL and the CuBLAS libraries. I experimentally showed that the utilization of GPU can be improved by parallel execution of multiplications, which is supported by the CUDA environment and also possible in the DMRG application.

To accelerate the projection operation, I proposed a hybrid implementation where multiplications are distributed between the available computing architectures. To improve the utilization of GPU during matrix multiplications, I designed a new scheduling algorithm supporting two different strategies. I created a single-threaded scheduling strategy for large matrices, in which case one multiplication can utilize all the GPU cores, and a multi-threaded strategy for small matrices, in which case relatively more memory is available and the parallel execution is advantageous. In the single-threaded strategy, multiplications are only scheduled to overlap the communications and the computations, however, for the multi-threaded strategy, I designed a complex algorithm, which also takes the limitations of CUDA kernel scheduling into consideration.

On the high-end K20 GPU, the parallel kernel execution significantly (44%) accelerated the multiplication of smaller matrices (range of 400-600), however, the total runtime of the algorithm slightly decreased (5%) as the average matrix size was larger in the investigated models. In practice, more complex models are investigated containing several symmetries which decrease the average matrix size and anticipate a higher speed-up.

DOI:10.15774/PPKE.ITK.2014.001

II.2 I developed a new algorithm for GPU to significantly increase the performance of the computation of the extremely asymmetric matrix-vector multiplications used in the DMRG algorithm. [2]

To accelerate the extremely asymmetric matrix-vector operations composing the second most time-consuming part of the DMRG algorithm, I designed a hybrid acceleration, in which the workload is distributed according to the available GPU memory and the performance capabilities of the two architectures. To improve the performance of the GPU part, I proposed a new algorithm to compute the transposed matrix-vector multiplication in CUDA environment, which outperformed the NVidia CuBLAS library by 4-5 times in the DMRG use-cases, where the number of rows of the matrix was in the range of 1-24. Overall acceleration of the matrix-vector operations including the data transfer as well reached approximately 2.4 times speed-up.

7.2. Új tudomanyos eredmenyek (in Hungarian)

A disszertáció két téziscsoport köré szerveződik. Az első téziscsoport lokálisan vezérelt *aritmetikai egységek* adatfolyam gráf leírásból történő generálásával foglalkozik. A második téziscsoport tárgya pedig a DMRG algoritmus első modern párhuzamos számítógép-architektúrákon történő megvalósítása.

I. Tézis. Magas működési frekvenciájú lokális vezérlést terveztem szinkron adatfolyam gráfok FPGA-n történő implementációjához, és módszert adtam a lokálisan vezérelt egységek számának és struktúrájának meghatározására a sebesség és a felület figyelembevételével.

I.1. Megterveztem és megvalósítottam egy lokálisan elosztott vezérlési módot, mely segítségével elkerülhetőek a globális vezérlőjelek és a vezérlés működési frekvenciája a felület mérsékelt növekedése árán növelhető. [3]

Kísérletileg megmutattam, hogy a szinkron adatfolyam gráfokból generált aritmetikai egységek esetén (pl. parciális differenciálegyenletek numerikus megoldása) a globális vezérlő jelek visszafogják az aritmetikai egység működési frekvenciáját, ha a ki- és bemenetek száma meghalad egy határértéket. Megterveztem és megvalósítottam egy

DOI:10.15774/PPKE.ITK.2014.001

lokálisan elosztott vezérlési módot, amely lehetővé teszi, hogy a lassú globális vezérlést elkerüljük a felület mérsékelt növekedése mellett.

Megmértem a tervezett vezérlés működési frekvenciáját a műveletvégző egységek nélkül, különböző számú ki- és bemenet esetén, hogy a vezérlés sebessége és a ki- és bemeneteinek száma között fennálló kapcsolatot meghatározzam. A nagy teljesítményű számításokhoz tervezett Virtex-6 FPGA esetén a maximálisan 10 ki- és bemenetet kezelő vezérlés 510 MHz frekvenciát ért el, amely körülbelül 20%-os gyorsulás a 20 ki-és bementet kezelő referencia esethez képest. A méréssel igazoltam, hogy a ki- és bemenetek számának korlátozásosával a vezérlés sebessége a szükséges mértékig növelhető, és lényegesen meghaladja a 450 MHz működési frekvenciát, melyet az áramkör többi része esetén (pl. lebegőpontos műveletvégzők) feltételezhetünk. A vezérlés esetén érdemes magasabb frekvenciát kitűzni célként, mivel gyakorlatban, amikor a műveletvégző egységek is megvalósításra kerülnek, jóval alacsonyabb működési frekvencia érhető el.

Optimalizálási feladatot terveztem az áramkör lokálisan vezérelt komponenseinek meghatározására, amely particionálja az adatfolyam gráfot, ha a ki- és bemenetek száma meghaladja a gyors működéshez szükséges határértéket. A létrejövő partíció osztályok egymástól függetlenül vezérelhetőek, az osztályok közötti szinkronizációt pedig FIFO (First-In-First-Out) bufferek biztosítják, melyek növelik a felhasznált konfigurálható logikai egységek számát (az áramkör felületigényét). A tervezett optimalizálási feladat célja, hogy a lokális vezérlés miatt kialakuló felületnövekedést minimalizálja a gyors működési frekvenciát biztosító particionálási feltételek mellett.

I.2. Kifejlesztettem egy mohó particionáló algoritmust, amelyik felülmúlta az egyik leggyakrabban használt korszerű particionáló eljárást a megfogalmazott optimalizálási feladat esetén. [4]

Megmutattam egy komplex áramlástani feladat aritmetikai egységének a megvalósításán keresztül, hogy a particionálás során megfogalmazható célkitűzések önmagukban nem elégségesek a magas órajel eléréséhez, és a generált áramkör elhelyezhetőségét is figyelembe kell venni. Terveztem egy mohó particionáló algoritmust, amely figyelembe veszi az elhelyezhetőség szempontjait, és segítséget nyújt a műveletvégző egység magas szintű manuális elhelyezéséhez. A mohó algoritmust és a Xilinx fejlesztőkörnyezet nyújtotta manuális elhelyezési lehetőségeket felhasználva 370 MHz mű-

DOI:10.15774/PPKE.ITK.2014.001

ködési frekvenciát értem el egyszeres pontosság mellett, felülmúlva a korszerű hMetis [53] algoritmust körülbelül 13%-kal. A generált áramkör manuális elhelyezés nélkül 328 MHz-et ért el egyszeres pontosság, és 296 MHz-et dupla pontosság esetén.

I.3. Kifejlesztettem egy új particionáló algoritmust, amely mind a particionálásbeli, mind az elhelyezésbeli szempontokat figyelembe veszi, és magas működési frekvenciát eredményez manuális elhelyezés nélkül is. [1, 5–7]

Egy új, a magas szintű szintézis során alkalmazható megközelítést javasoltam, mely a hagyományos, egymásra épülő lépésekből álló megközelítéssel szemben már a particionálás során figyelembe veszi az elhelyezési szempontokat. Terveztem egy szimulált lehűtésre épülő algoritmust a javasolt megközelítés bemutatására. Az algoritmus működését két komplex áramlástani példán keresztül is megvizsgáltam, megmérve a maximális működési frekvenciát a ki- és bementek számának függvényében. A nem particionált áramkörhöz képest mért legnagyobb gyorsulást (15-25%) akkor kaptam, amikor a partíció osztályok ki- és bemeneteinek a száma kisebb volt mint 9, illetve 10. Mindkét áramlástani példa körülbelül 320-325 MHz működési frekvenciát ért el dupla pontosság esetén.

II. Tézis. Az első hibrid CPU-GPU architektúrán megvalósított DMRG (Density Matrix Renormalization Group) implementáció hatékonyságának javítására ütemezőt terveztem a legidőigényesebb lépés mátrix-mátrix szorzásaihoz, és egy aszimmetrikus mátrix-vektor szorzó algoritmust dolgoztam ki GPU-hoz.

Elemeztem az algoritmus futásidejét, és az iteratív diagonalizáló eljárás (Davidson) projekciós lépését találtam a legidőigényesebbnek, amely kiszámolható egy sor sűrű mátrix-mátrix szorzás segítségével. Megvizsgálva a GPU és az FPGA architektúrák teljesítményét a sűrű mátrix-mátrix szorzásokra nézve azt találtam, hogy kellően nagy méretű mátrixok esetén a művelet mindkét architektúrán jó kihasználtság mellett elvégezhető, ugyanakkor teljes kihasználtságot feltételezve a GPU architektúra körülbelül 5-ször nagyobb teljesítményre képes, mint az FPGA. A CUDA környezet segítségével elkészítettem az algoritmus hibrid GPU-CPU implementációját, amely az algoritmus első modern párhuzamos architektúrán történő gyorsítása. A hibrid megoldás körülbelül 3.5-szörös sebességnövekedést ért el a csak CPU-t használó verzióhoz képest.

DOI:10.15774/PPKE.ITK.2014.001

II.1. Új ütemező algoritmust terveztem a DMRG legidőigényesebb lépéséhez, a projekciós műveletet leíró mátrix-mátrix szorzásokhoz, amely alkalmazkodva a GPU architektúra limitációihoz biztosítja a GPU magas kihasználtságát változó méretű mátrixok esetén. [8, 2]

Megvizsgáltam a projekciós operáció mátrix-mátrix szorzásaiban résztvevő mátrixok méretét a Heisenberg és a Hubbard modell esetén, melyek különböző számú szimmetriát tartalmaztak. Az eredményeket kiértékelve azt találtam, hogy a mátrix méretek széles skálán változnak mind az algoritmus iterációi között, mind magukon az iterációkon belül, és a mátrixok átlagos mértét nagyban befolyásolja a választott fizikai modell és a modellben szereplő szimmetriák száma.

Mérésekkel igazoltam a választott architektúrák teljesítménye és a mátrixok mérete közötti összefüggést a CPU esetén az MKL, a GPU esetén pedig a CuBLAS függvénykönyvtár segítségével. A méréseim alapján a GPU kihasználtsága hatékonyan növelhető több művelet párhuzamos végrehajtásával, amelyet mind a CUDA keretrendszer, mind a DMRG algoritmuson belüli alkalmazás lehetővé tesz.

A projekciós művelet gyorsítására egy hibrid megoldást javasoltam, melyben a szorzás műveletek szétosztásra kerülnek az elérhető számító egységek között. A GPU kihasználtságának javítására a GPU-n végrehajtandó műveletekhez egy olyan új ütemező algoritmust terveztem, amely két stratégia szerint képes a műveleteket ütemezni. Az egyszálú futtatáshoz szánt ütemezési stratégiát a nagyobb mátrixokhoz terveztem, amelyeknél a GPU kihasználtsága egy művelet végrehajtása során is megfelelő. Ebben a stratégiában a műveletek sorrendjét csak az határozza meg, hogy a kommunikáció és a munkavégzés átlapolható legyen. A többszálú stratégiát a kisebb mátrixokhoz terveztem, amelyek esetén relative több GPU memória áll rendelkezésre, de a párhuzamos műveletvégzés indokolt. A többszálú stratégiához tartozó ütemezés megvalósítására egy olyan algoritmust terveztem, amely nem csak a kommunikáció és a munkavégzés átlapolását hanem a CUDA kernelek ütemezésének a korlátait is figyelembe veszi.

A felsőkategóriás K20 GPU esetén a többszálú stratégia a kisebb méretű (400-600) mátrixok szorzását jelentős mértékben (44%) gyorsította, ugyanakkor az algoritmus teljes futási idejét tekintve csak mérsékelt (5%) gyorsulást eredményezett, mivel az alkalmazott modellekben viszonylag nagy mátrixok szerepeltek. A gyakorlatban összetettebb modellek is előfordulnak, melyekben több szimmetria szerepel, ami kisebb átlagos mátrix méretet és nagyobb sebességnövekedést vetít előre.

DOI:10.15774/PPKE.ITK.2014.001

II.2. Új algoritmust fejlesztettem GPU-ra, mely jelentősen megnövelte a GPU teljesítményét a DMRG-ben szereplő speciális, aszimmetrikus mátrix-vektor szorzások számítása során. [2]

Egy hibrid gyorsítást terveztem a DMRG algoritmus második legidőigényesebb részét képező szélsőségesen aszimmetrikus mátrix-vektor szorzásokhoz, amelyben az elvégzendő számítás az elérhető szabad GPU memória és az architektúrák teljesítménybeli különbsége alapján kerül szétosztásra. A GPU-ra jutó rész hatékonyságának növelésére egy új algoritmust terveztem a transzponált mátrix-vektor szorzás megvalósítására CUDA környezetben, amelyik a DMRG-s alkalmazás során, ahol a mátrix sorainak száma 1 és 24 között változott, 4-5-ször gyorsabbnak bizonyult, mint az NVidia CuBLAS könyvtár függvényei. A bemutatott gyorsítást a Heisenberg és a Hubbard modell esetén mérésekkel ellenőriztem, és a CPU-GPU kommunikációt is figyelembe véve körülbelül 2.4-szer gyorsabb aszimmetrikus mátrix-vektor szorzásokat eredményezett az NVidia K20 esetén.

7.2. Application of the Results

Two different types of computationally intensive problems have been researched to investigate the design methodology of the acceleration and to give a high-performance implementation on parallel architectures. Each problem was accelerated via a different architecture, and the results of the investigation were summarized in different thesis groups.

The design methodology proposed in Thesis 1 can be applied during any type of complex AU design when the AU has a significant number of I/Os and the performance takes priority over the area requirements. In my research, the AU design was motivated by the numerical solution of different conservation laws via the FVM discretization, however, other applications require complex AU design as well, e.g. Monte Carlo experiments requiring the computation of an expression with a lot of input variables.

Numerical solution of conservation laws was successfully demonstrated on FPGAs in case of simulation of CFD [1], electromagnetics [95] or seismic waves [96]. Areas profiting from the acceleration of these simulations include automotive, aircraft and wind power industries, circuit design and seismology.

The idea to feedback the high-level floorplan information to high-level circuit design can also be generalized. In the proposed methodology, the partitioning of the

DOI:10.15774/PPKE.ITK.2014.001

FPUs can be altered freely to find a favorable floorplan, however, in theory, any free design parameter could be tuned in a similar way. The proposed methodology can be integrated into high-level synthesis tools at the AU generation step or at other parts of the compilation process where a free parameter shall be optimized for speed.

The results of Thesis 2 were primarily applied in the GPU implementation of the DMRG algorithm, however, they can be used in further applications where similar challenges occur. The presented scheduling of matrix-matrix multiplications can be applied in Tensor Network (TN) methods [97], which compose a broader class of algorithms including DMRG as well, while the proposed kernel for asymmetric matrix-vector multiplication can be applied in Davidson implementations frequently used in quantum chemistry (e.g. [98]).

As the DMRG algorithm is one of the leading tools to study the low energy physics of strongly correlated quantum systems exhibiting chain-like entanglement structure, it can be applied to simulate anisotropic materials (e.g. polymers [99]) or to describe accurately the electronic structure of open d shell molecules [100]. Furthermore, the interacting system of atoms trapped in an optical lattice, proposed as physical implementation of quantum computer, is also tractable via DMRG [79].

References

Journal Publications of the Author

[1] Z. Nagy, C. Nemes, A. Hiba, Á. Csík, A. Kiss, M. Ruszinkó, and P. Szolgay, "Accelerating unstructured finite volume computations on field-programmable gate arrays", *Concurrency and Computation: Practice and Experience*, vol. 26, no. 3, pp. 615–643, 2014.

[2] C. Nemes, G. Barcza, Z. Nagy, Ö. Legeza, and P. Szolgay, "The density matrix renormalization group algorithm on kilo-processor architectures: implementation and trade-offs", *Computer Physics Communications*, 2014. DOI: 10.1016/j.cpc.2014.02.021.

Conference Publications of the Author

[3] C. Nemes, Z. Nagy, M. Ruszinkó, A. Kiss, and P. Szolgay, "Mapping of high performance data-flow graphs into programmable logic devices", in *Proceedings of the 2010 International Symposium on Nonlinear Theory and its Applications*, 2010, pp. 99–102.

[4] C. Nemes, Z. Nagy, and P. Szolgay, "Efficient mapping of mathematical expressions to fpgas: exploring different design methodologies", in *Circuit Theory and Design (ECCTD), 2011 20th European Conference on*, 2011, pp. 717–720.

[5] C. Nemes, Z. Nagy, and P. Szolgay, "Automatic generation of locally controlled arithmetic unit via floorplan based partitioning", in *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*, 2012, pp. 1–5.

[6] Z. Nagy, C. Nemes, A. Hiba, A. Kiss, A. Csik, and P. Szolgay, "Fpga based acceleration of computational fluid flow simulation on unstructured mesh geometry", in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 128–135.

DOI:10.15774/PPKE.ITK.2014.001

[7] Z. Nagy, C. Nemes, A. Hiba, A. Kiss, Á. Csík, and P. Szolgay, “Accelerating unstructured finite volume solution of 2-d euler equations on fpgas”, in *Conference on Modelling Fluid Flow (CMFF'12)*, 2012.

[8] C. Nemes, G. Barcza, Z. Nagy, Ö. Legeza, and P. Szolgay, “Implementation trade-offs of the density matrix renormalization group algorithm on kiloprocessor architectures”, in *Circuit Theory and Design (ECCTD), 2013 21th European Conference on*, 2013, pp. 100–104.

Related publications

[9] Z. Nagy, Z. Vörösházi, and P. Szolgay, “Emulated digital cnn-um solution of partial differential equations”, *International Journal of Circuit Theory and Applications*, vol. 34, no. 4, pp. 445–470, 2006.

[10] S. R. White, “Density matrix formulation for quantum renormalization groups”, *Phys. Rev. Lett.*, vol. 69, pp. 2863–2866, 19 1992.

[11] Ö Legeza, R. Noack, J. Sólyom, and L. Tincani, “Applications of quantum information in the density-matrix renormalization group”, in *Computational Many-Particle Physics*, ser. Lecture Notes in Physics, vol. 739, Berlin Heidelberg: Springer-Verlag, 2008.

[12] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, “Toward a software infrastructure for the cyclops-64 cellular architecture”, *High Performance Computing Systems and Applications, Annual International Symposium on*, vol. 0, p. 9, 2006.

[13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, “Introduction to the cell multiprocessor”, *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, Jul. 2005.

[14] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: a many-core x86 architecture for visual computing”, in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08, Los Angeles, California: ACM, 2008, 18:1–18:15.

[15] R. Rahman, *Intel® Xeon Phi™ Coprocessor Architecture and Tools*. Apress, 2013.

[16] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, “On-chip interconnection architecture of the tile processor”, *IEEE Micro*, vol. 27, no. 5, pp. 15–31, Sep. 2007.

DOI:10.15774/PPKE.ITK.2014.001

[17] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store", in *Green Computing Conference and Workshops (IGCC), 2011 International*, 2011, pp. 1–8.

[18] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: survey and challenges", *Found. Trends Electron. Des. Autom.*, vol. 2, no. 2, pp. 135–253, Feb. 2008.

[19] *NVidia Kepler GK110 Arhictecture Whitepaper*, www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, NVIDIA Corp, 2013.

[20] J. D. Anderson, *Computational Fluid Dynamics - The Basics with Applications*. McGraw Hill, 1995, ISBN: ISBN 2-930389-07-9.

[21] T. J. Chung, *Computational Fluid Dynamics*. Cambridge University Press, 2002.

[22] R. J. LeVeque, *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[23] S. Kocsárdi, Z. Nagy, A. Csík, and P. Szolgay, "Simulation of 2D inviscid, adiabatic, compressible flows on emulated digital CNN-UM", *International Journal on Circuit Theory and Applications*, vol. 37, no. 4, pp. 569–585, 2009. DOI: DOI:10.1002/cta.565.

[24] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse matrix test problems", *ACM Trans. Math. Softw.*, vol. 15, pp. 1–14, 1 1989, ISSN: 0098-3500.

[25] N. Gibbs, W. Poole, and P. Stockmeyer, "An algorithm for reducing the bandwidth and profile of sparse matrix", *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 236–250, 1976.

[26] D. Matzke, "Will physical scalability sabotage performance gains?", *Computer*, vol. 30, no. 9, pp. 37–39, 1997.

[27] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen, "Directed hypergraphs and applications", *Discrete Appl. Math.*, vol. 42, no. 2-3, pp. 177–201, Apr. 1993.

[28] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey", *Integration, the VLSI journal*, vol. 19, no. 1, pp. 1–81, 1995.

[29] M. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[30] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *The Bell system technical journal*, vol. 49, no. 1, pp. 291–307, 1970.

[31] A. Kahng, J. Lienig, I. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, 2011, ISBN: 0133016153.

DOI:10.15774/PPKE.ITK.2014.001

[32] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, *Handbook of Algorithms for Physical Design Automation*, 1st. Boston, MA, USA: Auerbach Publications, 2008.

[33] B. Hendrickson and T. G. Kolda, “Graph partitioning models for parallel computing”, *Parallel Computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[34] C. M. Fiduccia and R. M. Mattheyses, “A linear-time heuristic for improving network partitions”, in *Proceedings of the 19th Design Automation Conference*, ser. DAC ’82, Piscataway, NJ, USA: IEEE Press, 1982, pp. 175–181.

[35] L. A. Sanchis, “Multiple-way network partitioning”, *IEEE Trans. Comput.*, vol. 38, no. 1, pp. 62–81, Jan. 1989.

[36] M. Sarrafzadeh, M. Wang, and X. Yang, *Modern placement techniques*. Springer, 2002.

[37] D. A. Papa and I. L. Markov, “Hypergraph partitioning and clustering”, *Approximation algorithms and metaheuristics*, pp. 61–1, 2007.

[38] M. Fiedler, “Algebraic connectivity of graphs”, *Czechoslovak Mathematical Journal*, vol. 23, no. 2, pp. 298–305, 1973.

[39] K. M. Hall, “An r-dimensional quadratic placement algorithm”, *Management Science*, vol. 17, no. 3, pp. 219–229, 1970.

[40] C. J. Alpert, A. B. Kahng, and S.-Z. Yao, “Spectral partitioning with multiple eigenvectors”, *Discrete Applied Mathematics*, vol. 90, no. 1, pp. 3–26, 1999.

[41] G. Seber, “Multivariate observations john wiley & sons”, *New York*, 1984.

[42] S. Kirkpatrick and M. Vecchi, “Optimization by simmulated annealing”, *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[43] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, “Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning”, *Operations research*, vol. 37, no. 6, pp. 865–892, 1989.

[44] P. J. Van Laarhoven and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[45] S. T. Barnard and H. D. Simon, “Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems”, *Concurrency: Practice and experience*, vol. 6, no. 2, pp. 101–117, 1994.

[46] B. Hendrickson and R. W. Leland, “A multi-level algorithm for partitioning graphs.”, *SC*, vol. 95, p. 28, 1995.

[47] B. Hendrickson and R. Leland, “The chaco user’s guide: version 2.0”, Technical Report SAND95-2344, Sandia National Laboratories, Tech. Rep., 1995.

[48] G. Karypis and V. Kumar, “Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0”, 1995.

DOI:10.15774/PPKE.ITK.2014.001

[49] F. Pellegrini and J. Roman, "Scotch: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs", in *High-Performance Computing and Networking*, Springer, 1996, pp. 493–498.

[50] C. Walshaw and M. Cross, "Mesh partitioning: a multilevel balancing and refinement algorithm", *SIAM Journal on Scientific Computing*, vol. 22, no. 1, pp. 63–80, 2000.

[51] A. Trifunovic and W. J. Knottenbelt, "Parkway 2.0: a parallel multilevel hypergraph partitioning tool", in *Computer and Information Sciences-ISCIS 2004*, Springer, 2004, pp. 789–800.

[52] K. Devine, B. Hendrickson, E. Boman, M. S. John, C. Vaughan, and W. Mitchell, "Zoltan: a dynamic load-balancing library for parallel applications; user's guide", *Sandia National Laboratories Tech. Rep*, 1999.

[53] G. Karypis and V. Kumar, "HMETIS 1.5: A Hypergraph Partitioning Package", Department of Computer Science, Tech. Rep., 1998. [Online]. Available: `http://www-users.cs.umn.edu/~karypis/metis`.

[54] G. Karypis, K. Schloegel, and V. Kumar, "Parmetis: parallel graph partitioning and sparse matrix ordering library", *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[55] A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard cell vlsi circuits", *IEEE Transactions on Computer-Aided Design*, vol. 4, no. 1, pp. 92–98, 1985.

[56] R. Van Driessche and D. Roose, "A spectral algorithm for constrained graph partitioning i: the bisection case", *TW Reports*, p. 28, 1994.

[57] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures", *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 11, no. 2, pp. 109 –125, 1981.

[58] I. G. Tollis, G. Di Battista, P. Eades, and R. Tamassia, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998, ISBN: 0133016153.

[59] P. Eades and N. C. Wormald, "Edge crossings in drawings of bipartite graphs", *Algorithmica*, vol. 11, pp. 379–403, 4 1994, ISSN: 0178-4617.

[60] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz - Open Source Graph Drawing Tools", *Graph Drawing*, pp. 483–484, 2001.

[61] P. R. Panda, "Systemc: a modeling platform supporting multiple design abstractions", in *Proceedings of the 14th international symposium on Systems synthesis*, ser. ISSS '01, Montreal, P.Q., Canada: ACM, 2001, pp. 75–80.

DOI:10.15774/PPKE.ITK.2014.001

[62] B. Dezs, A. Jüttner, and P. Kovács, “Lemon - an open source c++ graph template library”, *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 5, pp. 23–45, Jul. 2011, ISSN: 1571-0661.

[63] *Xilinx CORE Generator System*, `http://www.xilinx.com/tools/coregen.htm`, Xilinx Inc, 2013.

[64] *Xilinx ISE Design Suite 13.1*, `http://www.xilinx.com/products/design-tools/ise-design-suite/`, Xilinx Inc, 2013.

[65] *Xilinx PlanAhead User Guide 13.1*, `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/PlanAhead_UserGuide.pdf`, Xilinx Inc, 2013.

[66] R. M. Noack and S. R. Manmana, “Diagonalization and Numerical Renormalization-Group-Based Methods for Interacting Quantum Systems”, *AIP Conf. Proc.*, vol. 789, pp. 93–163, 1 2004.

[67] U. Schollwöck, “The density-matrix renormalization group”, *Rev. Mod. Phys.*, vol. 77, pp. 259–315, 1 2005.

[68] G. Hager, E. Jeckelmann, H. Fehske, and G. Wellein, “Parallelization strategies for density matrix renormalization group algorithms on shared-memory systems”, *Journal of Computational Physics*, vol. 194, no. 2, pp. 795 –808, 2004.

[69] G. Alvarez, “Implementation of the su(2) hamiltonian symmetry for the {dmrg} algorithm”, *Computer Physics Communications*, vol. 183, no. 10, pp. 2226 – 2232, 2012, ISSN: 0010-4655.

[70] G. K.-L. Chan, “An algorithm for large scale density matrix renormalization group calculations”, *The Journal of chemical physics*, vol. 120, p. 3172, 2004.

[71] Y. Kurashige and T. Yanai, “High-performance ab initio density matrix renormalization group method: applicability to large-scale multireference problems for metal compounds”, *J. Chem. Phys.*, vol. 130, 23 2009.

[72] S. Yamada, T. Imamura, and M. Machida, “Parallelization design on multi-core platforms in density matrix renormalization group toward 2-d quantum strongly-correlated systems”, in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1–10.

[73] J. Rincón, D. J. García, and K. Hallberg, “Improved parallelization techniques for the density matrix renormalization group.”, *Computer Physics Communications*, vol. 181, no. 8, pp. 1346–1351, 2010.

[74] E. M. Stoudenmire and S. R. White, “Real-space parallel density matrix renormalization group”, *Phys. Rev. B*, vol. 87, p. 155 137, 15 2013.

DOI:10.15774/PPKE.ITK.2014.001

[75] J. Yu, H.-C. Hsiao, and Y.-J. Kao, “Gpu accelerated tensor contractions in the plaquette renormalization scheme”, *Computers & Fluids*, vol. 45, no. 1, pp. 55 –58, 2011, ISSN: 0045-7930.

[76] M. J. Cawkwell, E. J. Sanville, S. M. Mniszewski, and A. M. N. Niklasson, “Computing the density matrix in electronic structure theory on graphics processing units”, *Journal of Chemical Theory and Computation*, vol. 8, no. 11, pp. 4094–4101, 2012.

[77] F. Gebhard, *The Mott Metal-Insulator Transition: Models and Methods*. Springer, 1997.

[78] P. W. Anderson, *The Theory of Superconductivity in the High-Tc Cuprate Superconductors*. Princeton University Press, 1997.

[79] M. Lewenstein, A. Sanpera, V. Ahufinger, B. Damski, A. Sen(De), and U. Sen, “Ultracold atomic gases in optical lattices: mimicking condensed matter physics and beyond”, *Advances in Physics*, vol. 56, no. 2, pp. 243–379, 2007.

[80] A. I. Tóth, C. P. Moca, Ö Legeza, and G. Zaránd, “Density matrix numerical renormalization group for non-abelian symmetries”, *Phys. Rev. B*, vol. 78, p. 245 109, 24 2008.

[81] J. F. Cornwell, *Group Theory in Physics, An Introduction*. Academic Press, 1997.

[82] Ö. Legeza, J. Röder, and B. A. Hess, “Controlling the accuracy of the density-matrix renormalization-group method: the dynamical block state selection approach”, *Phys. Rev. B*, vol. 67, p. 125 114, 12 2003.

[83] *Intel Math Kernel Library 11.0*, `http://http://software.intel.com/en-us/intel-mkl`, 2013.

[84] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*. Manchester: Manchester University Press, 1992.

[85] B. Liu, “The simultaneous expansion for the solution of several of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices”, *Numerical Algorithms in Chemistry: Algebraic Method*, pp. 49–53, 1978.

[86] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, “Fpga based high performance double-precision matrix multiplication”, *International Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 322–338, 2010, ISSN: 0885-7458.

[87] G. G. Sleijpen and H. Van der Vorst, “A jacobi–davidson iteration method for linear eigenvalue problems”, *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 2, pp. 401–425, 1996.

DOI:10.15774/PPKE.ITK.2014.001

[88] E. R. Davidson, “The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices”, *Journal of Computational Physics*, vol. 17, no. 1, pp. 87 –94, 1975.

[89] M. Sadkane and R. Sidje, “Implementation of a variable block davidson method with deflation for solving large sparse eigenproblems”, English, *Numerical Algorithms*, vol. 20, no. 2-3, pp. 217–240, 1999.

[90] *Netlib repository*, `https://netlib.org`, 2013.

[91] *Cuda c programming guide 5.0*, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, NVIDIA Corp, 2013.

[92] *Kepler tuning guide 1.0*, `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html`, NVIDIA Corp, 2013.

[93] *CUBLAS library 5.0*, `https://developer.nvidia.com/cublas`, NVIDIA Corp, 2013.

[94] *CUDA library 5.0*, `http://www.nvidia.com/object/cuda`, NVIDIA Corp, 2013.

[95] J. Durbano and F. Ortiz, “Fpga-based acceleration of the 3d finite-difference time-domain method”, in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, 2004, pp. 156–163.

[96] H. Fu, W. Osborne, R. G. Clapp, O. Mencer, and W. Luk, “Accelerating seismic computations using customized number representations on fpgas”, *EURASIP J. Embedded Syst.*, vol. 2009, 3:1–3:13, Jan. 2009.

[97] J. I. Cirac and F. Verstraete, “Renormalization and tensor product states in spin chains and lattices”, *Journal of Physics A: Mathematical and Theoretical*, vol. 42, no. 50, p. 504 004, 2009.

[98] C. Vömel, S. Z. Tomov, O. A. Marques, A. Canning, L.-W. Wang, and J. J. Dongarra, “State-of-the-art eigensolvers for electronic structure calculations of large scale nano-systems”, *J. Comput. Phys.*, vol. 227, no. 15, pp. 7113–7124, Jul. 2008.

[99] W. Barford, *Electronic and Optical Properties of Conjugated Polymers*. Oxford University Press, 2005.

[100] G. Barcza, Ö. Legeza, K. H. Marti, and M. Reiher, “Quantum-information analysis of electronic states of different molecular structures”, *Phys. Rev. A*, vol. 83, p. 012 508, 1 2011.