University of Alberta

Parallel Electromagnetic Transient Simulation of Large-Scale Power Systems on Massive-threading Hardware

by

Zhiyin Zhou

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Energy Systems

Department of Electrical and Computer Engineering

Fall 2012
Edmonton, Alberta

To my parents and my families,

for their support as always.

Abstract

Electromagnetic transient (EMT) simulation is widely utilized in power system planning and design. Respecting the detail and complexity of the components models, the electromagnetic transient program (EMTP) demands significant computational power. Increasing with the scale of the system, this requirement has become so prominent that parallel programming techniques are urgently needed in large-scale power system EMT simulation. Improving upon the multithreading parallelism, massive-threading computing is one of the key developments that can increase the EMT computational capabilities substantially when the processing unit has enough hardware cores. Compared to the traditional central processing unit (CPU), the graphic processing unit (GPU) has many more cores with distributed memory which can offer higher data throughput.

This thesis describes the conception of the massive-threading parallel EMTP based on GPU for large-scale power systems using compute unified device architecture (CUDA). It defines a new fundamental program framework, relevant basic data structures and efficient data interfaces of the massive-threading parallel EMT simulator. The thesis proposes a series of massive-threading parallel modules for component models including unified linear passive elements module (ULPEM), universal line module (ULM) and universal machine module (UMM); and numerical methods, including Newton-Raphson iteration module (NRIM) and forward-backward substitution with LU factorization module (FBSLUM), used in the EMTP. Without the need for a trade-off between the system scale and the complexity of the component models, all parallel modules proposed above are detailed, universal and unified. In order to fully release the computing power of modern computer system, both data and instructions are based on 64-bit architecture, which guarantee the precision as well as extensibility of the program.

The developed MT-EMTP program has been tested on various large-scale power systems of up to 2458 three-phase buses with detailed component modeling. The simulation results and execution times are compared with a mainstream commercial software, EMTP-RV®, to show the improvement in performance with equivalent accuracy.

Acknowledgements

I would like to express my sincere thanks to my supervisor *Dr. Venkata Dinavahi* for his full support, encouragement, and guidance for the years throughout my research at the University of Alberta. His insightful guidance, passion and enthusiasm for the research has been an invaluable motivation for my life.

It is an honor for me to extend my gratitude to my M.Sc. committee members *Dr. Behrouz Nowrouzian* and *Dr. Amit Kumar (Mechanical Engineering)* for reviewing my thesis and providing invaluable comments. Special thanks go to my colleagues and friends at the RTX-Lab: *Yuan Chen*, *Song Wang*, *Jiadai Liu* and *Yifan Wang*.

Finally, financial help from NSERC, the University of Alberta, Government of the Province of Alberta for my living in Edmonton during these years is greatly appreciated.

Table of Contents

List of Tables

List of Figures

List of Acronyms

APIs	Application Programming Interfaces
BNA	Block Node Adjustment
CMIM	Compensation Method Interface Module
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random-Access Memory
ECC	Error-Correcting Code
EMT	ElectroMagnetic Transients
EMTP	ElectroMagnetic Transients Program
LU-FBSM	LU & Forward-Backward Substitution Module
GPGPU	General Purpose Computing on GPU
GPU	Graphic Processing Unit
MT-EMTP	Massive-Threading based EMTP
NLE	NonLinear Element
NMS	Node Mapping Structure
NRIM	Newton-Raphson Iteration Module
OS	Operation System
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
TDP	Thermal Design Power
ULPEM	Unified Linear Passive Elements Module
ULM	Universal Line Module
UMM	Universal Machine Module
VF	Vector Fitting

1 Introduction

Electromagnetic transients (EMT) are the temporary electromagnetic phenomena, such as changes of voltage, current and flux, in a short time slice caused by the excitation due to switching operation, fault, lightning strike, and other disturbances in power system [1]. Although they are short and fast, transients impact the stability and reliability of the power system significantly. For example, transients can damage component insulation, activate control or protective systems, and cause system interruption; thus studying and analyzing EMT play important roles in the planning, design, and operation of modern power systems. An EMT simulator numerically simulates them by using computer analytical models to illustrate these phenomena in detail. Nowadays, the use of EMT simulation tools is no longer restricted to specialized studies focused on analyzing the propagation of electromagnetic transients. Due to their versatility and breadth of modeling capability, EMT tools such as ATP [2], PSCAD/EMTDC® [3], EMTP-RV® [4] and etc., are routinely used in the planning, design and operation of power systems, to study dynamic phenomena over a wide frequency range— from steady-state studies such as load flow and harmonic analysis to high-frequency studies such as restrike overvoltages in gas insulated substations [5].

Along with modeling and application diversity, the size of the power system simulated by EMT tools has grown concomitantly [6]. It is not uncommon to be able to simulate in detail systems containing hundreds of buses using such tools. Nevertheless, the common characteristic of all existing EMT simulation tools is that they are single-thread sequential programs designed to run efficiently on single-core CPUs based on the x86 processor architecture. Throughout the 1990s and 2000s, the CPU clock speed steadily increased and memory costs decreased fueling a sustained increase in the speed of these programs. But now with the clock speed saturated around 3GHz due to chip power dissipation and fabrication constraints, the computer industry has transitioned to multi-core hardware ar-

chitectures in a CPU to improve overall processing performance, so that the performance of a processor is no longer only decided by the frequency.

1.1 Single-thread and Multi-thread Programming

In a traditional single-tasking operation system (OS), the CPU can only execute the instructions for one task at any point in time. The execution of code and access of data are serial and sequential since the program is single-threading. The conception of multi-threading comes up with a multi-tasking OS, which was considered much earlier than a multi-core hardware. Since the executing speed of a CPU core is much faster than peripherals, such as DRAM, hard drive and IO ports, multi-threading reduces time wasted waiting for low speed peripherals. Although still running on the single-core CPU, threads in a multi-threading program are concurrent by sharing CPU time and switching context, scheduled by the multi-tasking OS. The multi-threading program really achieved parallel execution only after the advent of multi-core CPU; on the other hand, a program can expoit the computing power of multi-core CPU only if it supports multi-threading. Therefore, executing a single-threading EMT program on a multi-core architecture is inefficient because the code is executed on a single core, one instruction after another in a homogeneous fashion, unable to exploit the full resource of the underlying hardware. The overall performance of the code can be severely degraded especially when simulating large-scale systems with high data throughput requirements. A multi-threading parallel code can provide substantial gain in speed and throughput over a single-threading code on a multi-core CPU. Even on single-core processor systems, multi-threading can add palpable performance improvement in most applications. The implementation of multi-threading, however, is not that natural because a problem is natively coupled and sequential in common. A serial data structure and algorithm are required to be redesigned to accommodate the multi-threading pattern.

1.2 Massive-Thread Programming

The idea of massive-threading is based on one of the most advantageous modern processor techniques— the many-core processor. The original motivation of many-core processor was to accelerate the 3D graphics for digital graphic processing. Under the high demand of powerful 3D graphics, the graphic processing unit (GPU), which is a specialized electronic circuit originally designed to manipulate the display and 2D/3D video processing in high-speed [7], was conceived in the 1990s as a hardware-accelerated 3D graphic processor. It normally contains many cores organized as multiple Streaming Multiprocessors (SMs) with a massive parallel pipeline, than a conventional CPU with just multiple cores. With the development of fast semiconductor material and the manufacturing technology in integrated circuit industry, more and more (up to thousands nowadays) cores are being inte-

grated into one chip. Trying to use the processing power of a GPU to relieve the increasing pressure of CPU in high compute-demanding tasks besides the normal graphic processing, general purpose computing on GPU (GPGPU) was proposed to turn the graphic processing power into general-purpose computing power [8, 9]. Researchers have already begun to exploit GPUs for various applications in multiple industries [10, 11], such as molecular dynamics, bio-informatics, computational fluid dynamics, finance, weather and atmosphere modeling, etc. In power systems the applications include data visualization, load flow computations and transient stability simulations [12–19].

However, unlike the cores in common multi-core CPU, the GPU cores are lightweight processors without complicated thread control, thus single instruction multiple data (SIMD) technique invented in the 1970s' vector supercomputers is applied widely in GPU for both graphic and general purpose computing, which can accelerate the procedure and data independent computations effectively. Since the GPU was designed for graphic applications natively, the GPU functions are difficult to be used in general purpose computing, which requires a computer engineer to have enough graphic processing knowledge and transfer a normal mathematic problem to a graphics problem, Several developing platforms are provided to help the GPGPU development, which will be given briefly as follows:

- CUDA™ (compute unified device architecture) offers a C-like language to develop GPGPU application on the GPU provided by NVIDIA®.
- OpenCL™ is an open framework for developing GPGPU programs, which can be executed across various platforms, supporting the GPUs of AMD®, INTEL® and NVIDIA®. Initiated by Apple® with the collaboration of a group of GPU providers, OpenCL is developed by a nonprofit organization Khronos Group™.
- DirectCompute, a part of DirectX® 11, is a set of application programming interfaces (APIs) that supports GPGPU with DirectX 11 supported GPUs on Windows® provided by Microsoft®.

All the above development platforms cover the detail hardware structures of GPUs and provide a relatively unique program interface, making it easier for software developers to massive-threading for their parallel programs.

1.3 Parallel Performance of a Program

With multi-threading based on a multi-core processor, a program can be accelerated by parallel computing. Then, how does one predict the overall performance of a multi-threading program and how do we measure the speedup of the parallel program relative to the serial counter part? Amdahl's law [20] answers the above questions of our EMT simulation problem, in which the workload is scaled. In following discussion, we assume all threads are running on enough real cores. The multi-threading based on a single-core

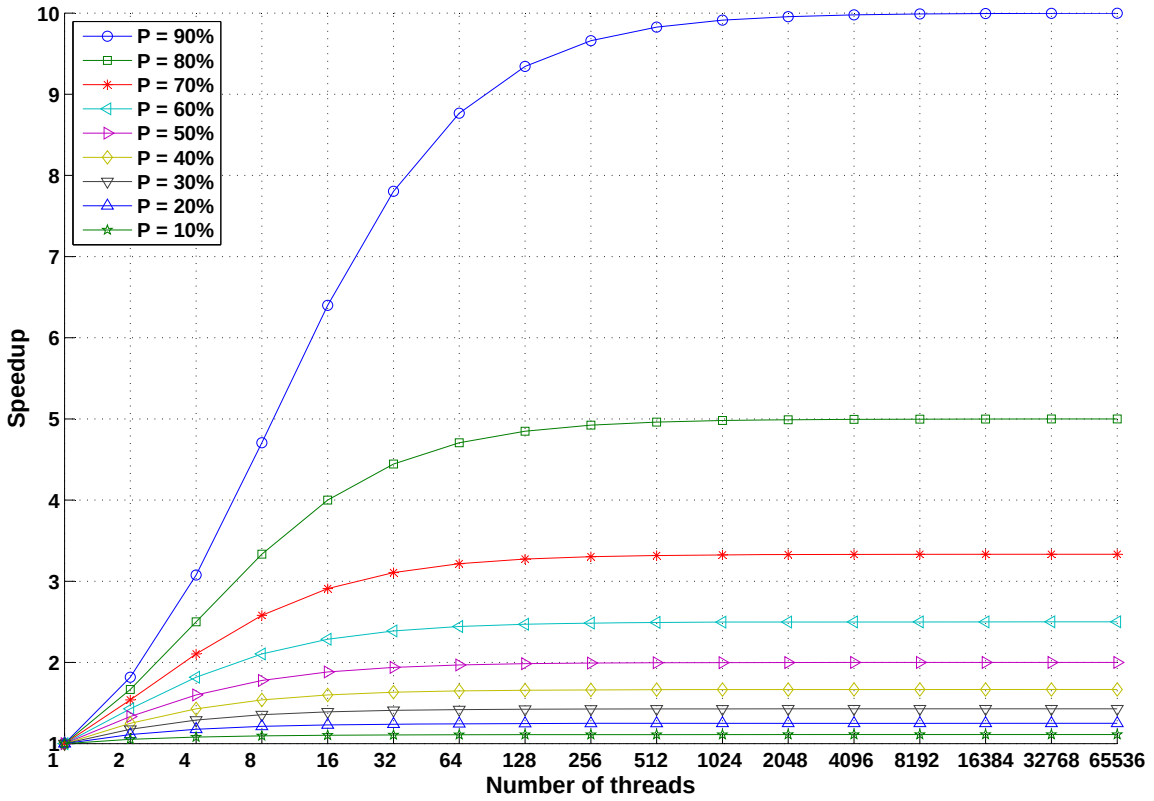

Figure 1.1: The Amdahl's Law

or fewer-core processor is not included. Under this assumption, the number of threads presents the number of work-flows executing in parallel. The speedup S with N threads is given as

$$S(N) = \frac{1}{(1-P) + P/N}, \tag{1.1}$$

where P is the proportion of the program which can be parallelized. As shown in Fig. 1.1, the limit of the speedup is decided by the proportion which cannot be parallelized instead of the number of threads in a scaled task. For example, if the parallel proportion is less than 50%, the overall speedup can never exceed 2. Thus, the processing method and the data structure which can increase the degree of parallelism are the linchpin in the eventual acceleration for a massive-threading program since there are supposed to be enough threads for the job. Moreover, Amdahl's Law shows that the speedup climbs fast with the increase in the number of threads at the beginning, but slows down and eventually saturate. Therefore, the granularity, which is defined as the amount of work of a single task in a parallel program [21], should be properly measured to achieved the balance between the computing resource and the computing performance [22].

1.4 Motivation for this work

In the mainstream EMT simulation software nowadays, there is no acceptable practical solution to simulate a realistic large-scale power system with detailed models without using reduced equivalent. It normally takes hours to simulate a power system with tens of thousands of buses completely on a single-core CPU [6]. Without multi-threading and massive-threading techniques, these software cannot derive any benefit from the extra cores even if they run on a multi-core CPU. A practical way to solve this problem without parallel computing is to reduce the system complexity. The simulation software will use detailed models for the focused parts of the large system, and the remainder parts of system are equivalent to reduced models which are obviously less accurate than using detailed models. However, it still takes several minutes even for the reduced equivalents to simulate a few milliseconds of transient behavior [6].

In order to meet the requirement of the endless growing complexity and size of modern power systems, Massive-threading is proposed in this thesis to evolve the EMT simulator as a leading-edge parallel technique. However, it is a significant challenge to implement massive-threading for EMT simulation due to the many-core processor's different hardware architecture and its cooperation with the CPU. As per the analysis of the parallel performance in Section 1.3, simply adding more threads cannot provide more improvement in speed— considering the cost of cooperation, the overall performance may even be jeopardized. Therefore, increasing the rate of parallelism of EMT program, such as scattering the data structures for the network solution and decoupling the process routing for the complicated detailed models, and decreasing the cost of the cooperation, such as reducing the data transfer between different processors and controlling the response delay of recurrent operations, are the most important approaches to the overall improvement in speed, since the modern commercial EMT software has already gained very high performance with the mature algorithms and full adaption on the CPU. The mind-set has to be changed from either the traditional single-threading or common multi-threading programming based on the CPU to massive-threading on a many-core processor.

This thesis focuses on this challenge to design a massive-threading EMT program (MT-EMTP) for large-scale power system using detailed component models based on GPU [23], which has native parallel many-core processing units and high-performance floating-point number processors, and leads the EMT simulation to a new direction of parallel computing. In the view of the author, it is the first time that the MT-EMTP is implemented using the most detailed models of power system components. The program is developed using CUDA 4 [24], a relatively mature and stable platform, with C++, which can be ported to other platforms conveniently.

1.5 Research Objectives

The objectives to be accomplished in developing the massive-threading parallel program for EMT simulation (MT-EMTP) are listed as follows:

- The unified passive elements module (ULPEM) for common linear power system components, such as resisters, capacitors, inductors and switches.
- The compensation method interface module (CMIM) for common non-linear power system components, such as surge protector, lightning arrestor and nonlinear RLC.
- The universal line module (ULM) for detailed modeling of transmission lines and cables.
- The universal machine module (UMM) for detailed modeling of synchronous machines.
- The forward-backward substitution with LU factorization module (LU-FBSM) for the solution of linear equations using the block node adjustment (BNA) to obtain a block diagonal pattern for the system admittance matrix, which is ideally suited for the GPU-based massive-threading parallel computing.
- The Newton-Raphson iteration module (NRIM) for the solution of nonlinear equations.
- The performance of the MT-EMTP is evaluated for accuracy, computational efficiency and scalability, using several large-scale test power systems, and compared with the EMTP-RV®.

1.6 Thesis Outline

This thesis consists of four chapters. Other chapters are outlined as follows:
Chapter 2 gives a general introduction to GPU architecture and CUDA abstraction, and also discusses some important design issues.
Chapter 3 describes the framework of the simulator; the compensation method interface; and the details of the massive-threading parallel modules, including ULPEM, ULM, UMM, NRIM and LU-FBSM.
Chapter 4 presents the experimental results for various large-scale test systems and comparison with EMTP-RV®.
Chapter 5 gives the conclusions and future work.

2

GPU Architecture and CUDA™ Abstraction

As the crux of a massive-threading parallel EMT simulation system, the GPU plays an undeniably important role in it. After experiencing many generations, the architecture of GPU has matured along with its performance. The Fermi™ architecture GPU was used for the development of the parallel EMT simulation system in this thesis. Along with the hardware architecture, the software architecture for the parallel programming of NVIDIA's GPU, the compute unified device architecture (CUDA™), has also progressed rapidly up to Version 4, which became the foundation of the software framework for the parallel EMT simulation system.

Table 2.1: GPU Specification

Chip	GF100
Die size	529 mm^2
Fabrication	40 nm
Transistors	3.2 billion
Number of SMs	16
Number of cores	512
L1 Cache	64 KB
L2 Cache	768 KB
Bus width	384 bit
Memory ECC	Supported
System interface	PCIe 2.0 $\times$ 16

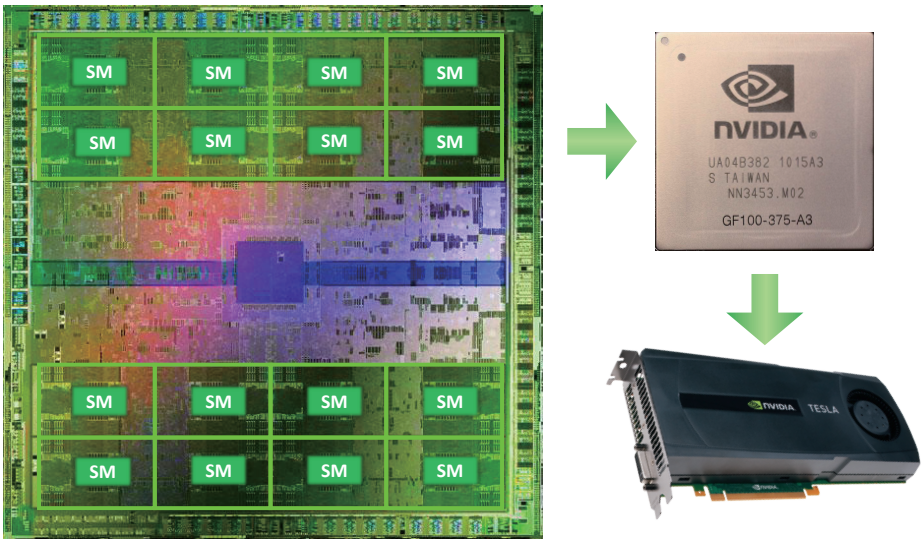

Figure 2.1: Die, chip and card for NVIDIA® Fermi™.

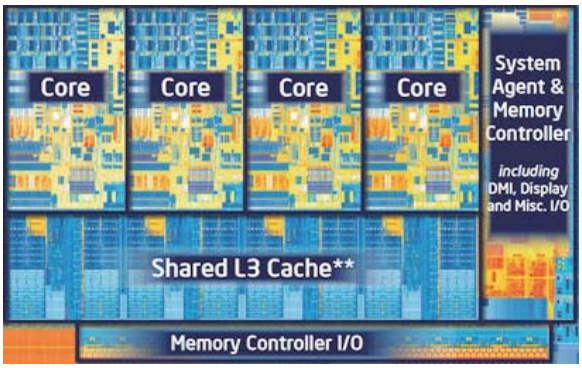

Figure 2.2: Die of Intel® Core™ I7.

2.1 GPU Architecture

As the 11th generation of NVIDIA's GeForce GPU and the 3rd generation CUDA supported product, Fermi™ (GF100) [25] has a lot more compute capability than its predecessors. According to the specification listed in Tab. 2.1, the GF100 has 3.2 billion transistors in the 40nm process on the 529 mm^2 die. There are 16 Streaming Multiprocessors (SMs) with 32 cores each; thus, there are 512 cores in total, in the GF100. It connects to the main system by PCIe 2.0 $\times$ 16 interface with a 384-bit wide bus. Fig. 2.1 shows the GPU card (C2050) made of the GF100 consisting of the many-core die. With 3rd generation SM, GF100 offers 8$\times$ faster double precision floating-point performance over its predecessor. Moreover, it has the true cache hierarchy, and each SM has 64 KB RAM with a configurable partitioning of shared memory and L1 cache to improve bandwidth and reduce latency. Additionally, benefitting from the NVIDIA GigaThread™ Engine, it can deal with the application context switching with 10$\times$ faster speed, and execute an application in concurrent kernels and

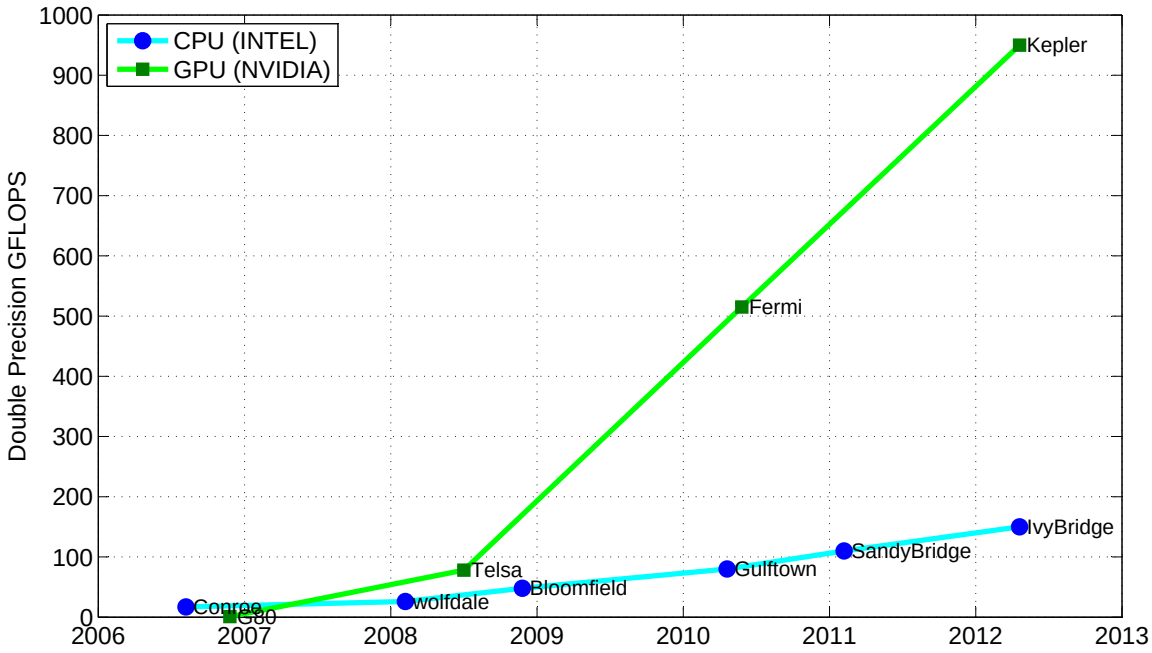

Figure 2.3: Computing performance comparison (CPU VS. GPU).

out of order threads. Last but not least, Fermi is the first GPU architecture supporting error-correcting code (ECC) to increase the data reliability during the computation.

Different from the most modern multi-core CPU, Intel® Core™ i7 whose die is shown in Fig. 2.2, which only has several serial cores, the GPU which has much more cores is optimized for throughput with explicit management of on-chip memory. Although one core of CPU has much more power than that of a GPU for a serial and random task, the GPU will release its marvelous power to show extreme speedup when the task can be parallelized into many threads and executed in a single instruction multiple data (SIMD) format.

Since the capability to process floating-point operations is an important aspect in modern computing as well as the EMTP simulation, floating-point operations per second (FLOPS) has been a key index to measure the performance of a compute system. Fig. 2.3 shows the comparison between CPU and GPU during recent years in double precision Giga-FLOPS computation [26]. Although both CPU and GPU increase their computing performance steadily, the increase of GPU's performance is much sharper than that of the CPU. The GPU used in this project (Fermi) has about 5 times GFLOPS than the mainstream CPU (SandyBridge), while clocking at half the speed of the CPU. Meanwhile, Fig. 2.3 shows a bright prospect of GPU computing performance as well.

The diagram in Fig. 2.4 shows the GPGPU computing system architecture. It consists of 14 SMs, and each SM is populated with 32 compute cores which share the registers, caches and dedicated memory inside the SM. There is a total of 64KB memory in each SM, which can be configured into 48KB shared memory and 16KM L1 cache, or reconfigured

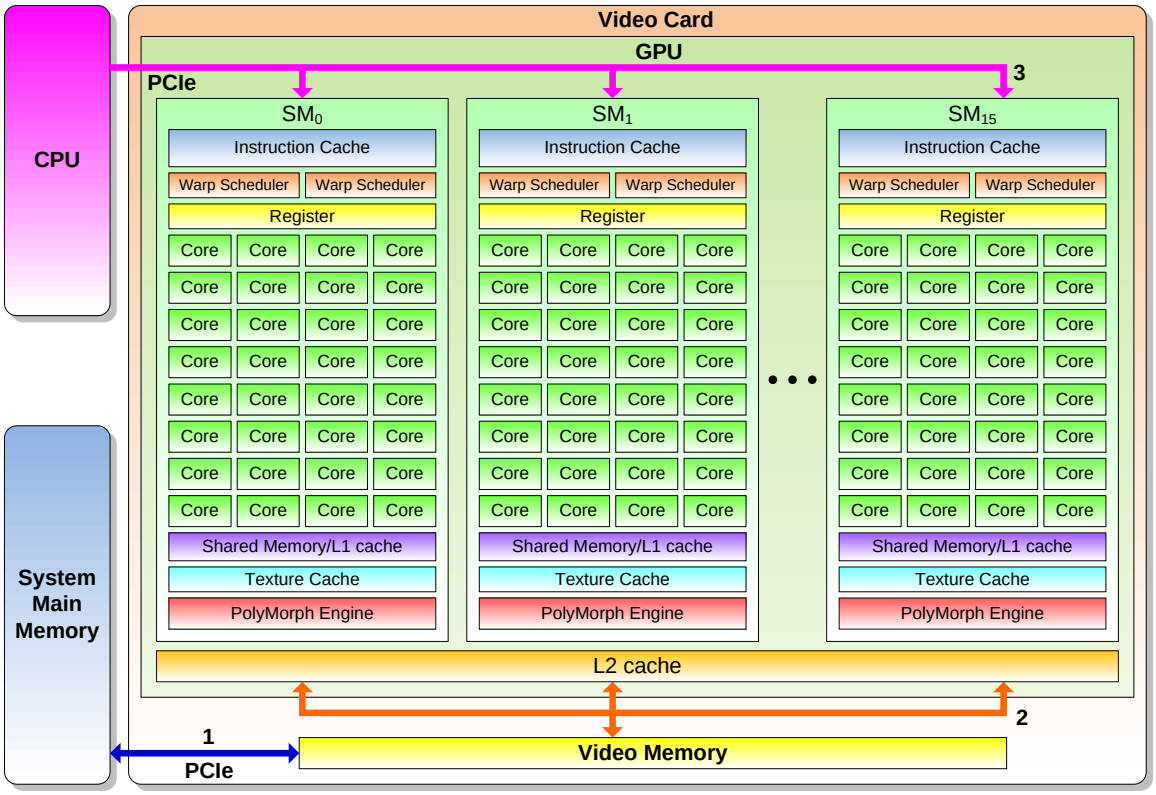

Figure 2.4: Computing system architecture: (1) data transmission between host and device, (2) data dispatched to/from many cores, (3) Control instruction dispatch.

into 16KB shared memory and 48KB L1 cache. 768KB unified L2 cache, which can be read and written by all clients, increases the throughput of off-chip memory (video memory) efficiently. Since the GPU is designed to work as a coprocessor, all data and instructions come from the CPU via the PCIe interface. Before computation, the raw data must be transferred into the 384-bit width GDDR5 video memory by path 1 in Fig. 2.4, which can offer maximum 16GB/s bi-direction bandwidth. And then, the data are distributed to every computing core via path 2 in Fig. 2.4, which can offer maximum 144GB/s bandwidth. Controlled by the instructions from path 3 in Fig. 2.4, the calculations are executed in parallel. Although the video card connects to the main system by PCIe 2.0 $\times$ 16 bus and the video memory running in 3GHz has 384-bit width interface, they can still not fulfill the 515.2GFLOPs double-precision peak floating point computing power of the GF100.

In order to make every core in the GPU work efficiently, enough data must be fed to catch up to the instruction cycles; without data input, the cores can only remain idle, thus reducing compute speed. Since the GPU has far more (hundreds) cores than the CPU, the data bandwidth requirement is increased tremendously. However, both the system main memory and the video memory cannot offer that ideal bandwidth; therefore, specific memory access routes have to be followed to reach the optimal speed. The data transmission

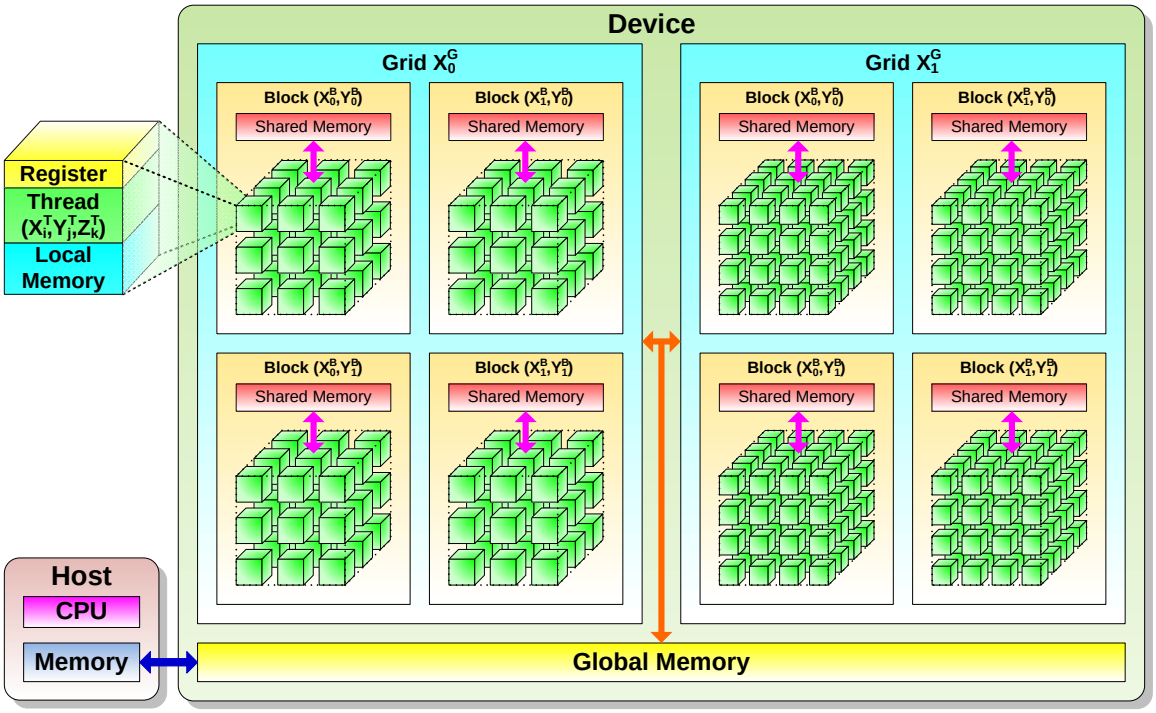

Figure 2.5: CUDA abstraction.

paths between the memories, i.e. path 1 and path 2 in Fig. 2.4, are the main bottlenecks of the architecture. As explained later, the proposed massive-threading parallel EMT modules are designed to minimize the use of these paths in order to maximize computational efficiency.

2.2 CUDA Abstraction

Compute Unified Device Architecture (CUDA) proposed by NVIDIA is the programming language used to implement a parallel program without dealing with the GPU assembly language. The NVIDIA GPU hardware are abstracted by CUDA architecture, so that the software developed by CUDA can execute in various CUDA GPUs (CUDA-supported GPUs) in spite of their different hardware specifications, such as the number of SMs per chip and cores per SMs. With C-like syntax called CUDA C and C-format API, CUDA supports major C/C++ features, such as pointer and class, to integrate with C/C++ program easily. Since CUDA has a relatively longer history that other GPGPU developing libraries, there are a group of special application libraries available, such as CUBLAS for linear algebra, CUFFT for Fast Fourier Transform and CURAND for random number generation. As well as the general abstraction, all hardware details of GPUs can still be configured and set by CUDA without missing the flexibility. The integrated development environment, multiple operation system platform support and extensive debugging tools also benefit the programmers and developers.

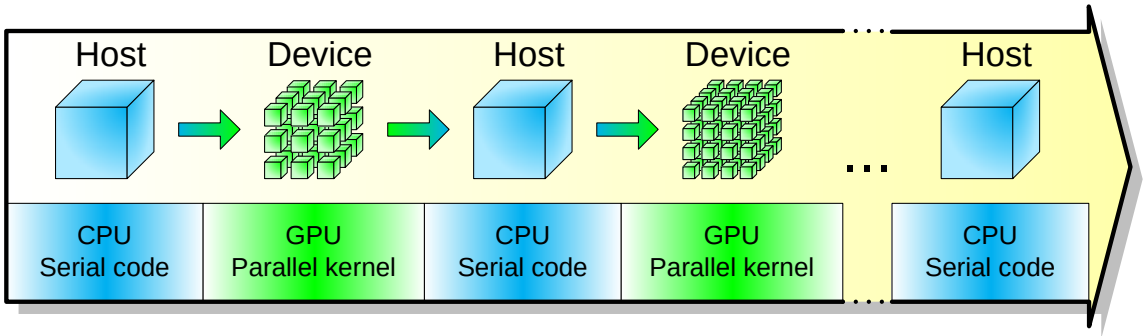

Figure 2.6: Heterogeneous Programming.

As shown in Fig. 2.5, the computing system is separated into two parts: CPU side as the *host*, on which the serial parts of the program run, and the GPU side as the *device*, on which the parallel parts of the program run, in the CUDA architecture. Since the GPU is a co-processor of the computing system, the basic idea of *heterogeneous* programming is to run the serial part and parallel part of a program on the CPU and GPU. In the CUDA architecture, the serial part is programmed by normal C/C++ code; the parallel part is programmed into a kernel by CUDA C. When the whole program is executed, it starts from the CPU, then the serial codes run on the host side, and the parallel kernels run the device side. The two sides execute their programs alternately until the end of execution, as shown in Fig. 2.6. Considering the Amdahls Law mentioned in chapter 1, extending the parallel part as much as possible is the critical path to gaining the maximum acceleration from GPU.

2.2.1 Thread hierarchy

Since there are millions of threads applied in the CUDA architecture, a CUDA thread hierarchy is designed to manage them. This is a 3-level hierarchy. As shown in Fig. 2.5, Grid, Block and Thread map to GPU, SM and Core respectively. However, the user need not be concerned with the actual number of GPUs, SMs and Cores, since the number of abstracted threads are automatically assigned to the physical cores in parallel or serial fashion. Therefore, even if the GPU used has fewer cores than what the CUDA program requires, the programmer can still claim the number of threads needed. The grid is the top level in the CUDA hierarchy, normally one GPU means one grid. Therefore, for a single GPU computing system, the index of grid, x^G, is always 1. On the other hand, a computing system can have multiple GPUs so that the device side of CUDA can maintain multiple grids as well. In one grid, there are many blocks which are divided in 3 groups (x,y,z). The index of block per grid, (x^B, y^B, z^B), is a 3-dimensional vector, with the maximum size of each dimension being 65525. Therefore, one grid can contain a maximum 281462092005375 (65535×65535×65535) blocks. Similar to blocks in grid, there are thousands of threads in

one block. The index of thread per block, (x^T, y^T, z^T), is also a 3-dimensional vector. Different from the maximum size of the block dimensions, at the thread-level the maximum sizes of x and y dimensions are 1024, and the maximum size of z dimension is 64. Therefore, the maximum index dimension of threads is 67108864 (1024×1024×64). However, the maximum number of thread in a block is limited to 1024. Under this restriction, if one dimension uses up all 1024 threads, the other dimensions have to be set to 0.

All threads in a block share the resources of the block , such as register, shared memory and cache. In order to avoid the conflict of using shared resources, all threads in each block can be synchronized by setting the synchronization point specifically, which acts a barrier where all threads in the block must wait before any is allowed to proceed. All threads inside a grid execute the same instruction simultaneously with multiple data input, which requires complete data independence and unified processing flow in the kernel. This is known as the single instruction multiple data (SIMD) format. On the other hand, the single instruction multiple thread (SIMT) enables the program with thread-level parallel code for independent, scalar threads [24], thereby allowing the GPU to handle multiple branches and operations in a single instruction. The developed parallel EMT component modules and the sparse linear solver utilize both the SIMD and SIMT concepts.

2.2.2 Memory hierarchy

Similar to the division of execution into host and device sides, the memory structure is also divided into these two sides. The host memory is on the host side; and the memories on the device side include global, shared and local memory. The host memory maps the system main memory of the computer system, which can only be accessed by the CPU. Since the GPU cannot access the system main memory, the threads on the device side cannot access the host memory. While, on the device side, each thread has its own register and local memory; each block has its dedicated shared memory, which has much lower access latency and wider bandwidth than others, used by all threads inside the block; all threads in the grid can access the global memory, which maps to the video memory on the video card, as shown in Fig. 2.5. Therefore, all data being processed by the GPU has to be first copied into the global memory on the device side. Then the associated data are transferred to the relevant memory and register belonging to the respective block and thread. The computations in each thread will not start until all data are ready.

Since the global memory is not on chip though it is on board, accessing it is relatively inefficient. The shared memory (much like an L1 cache), which is much faster than global memory, is offered for each block, which can be accessed by all threads in the block. Thus using this limited resource (48KB per block) wisely can effectively optimize the performance of the program. The advanced Fermi architecture (compute capability 2.0 and above) offers data cache (16KB per block) which requires a well organized data input to maximize access speed. In the global and shared memories, a memory address normally

can only be accessed once at the same time in an instruction cycle, especially for write operation. Therefore, simultaneous multiple and random memory access should be avoided in the CUDA kernel because not all threads can guarantee that their memory access is safe. Because there are numerous memory transactions before each parallel operation, the performance of memory throughput influences the overall performance significantly, sometime it even overwhelms the speed and number of cores of the GPU. It is obvious that all threads (cores) just keep idle before all data are prepared. Since all memory interfaces do not have ideal bandwidth and are predefined, the alternative way to optimize performance is to maintain the wide scale memory transactions as few as possible, which defines the keynote of the data structure design for the parallel EMT simulation.

2.2.3 Syntax extension

The function implementing the parallel code is named kernel in CUDA. CUDA C extends the syntax of standard C by defining the kernel. The notations,

__global__ and **__device__**,

are introduced to declare the types the variable and function. The global function can be called on the host side; and the device function can be called on the device side.

Before the global function is called, the configuration of threads is specified using a new execution configuration syntax,

<<< $\cdots$ **>>>**.

Inside the bracket, the dimension of blocks per grid, the dimension of threads per block and the amount of shared memory per block are declared explicitly. Based on this declaration, each thread executing the kernel has a unique thread ID within the kernel given by the built-in variables,

threadIdx, **blockDim** and **blockIdx**.

The thread ID of a block can be expressed as

`threadIdx.z*blockDim.y*blockDim.x + threadIdx.y*blockDim.x + threadIdx.x`.

The following sample code shows a kernel for adding two vectors `A` and `B` of size `N`, and for storing the result in the vector `C`:

```
//Kernel definition
__global__ void VecAdd(int N, float* A, float* B, float* C)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main()
{
    ...
    //Kernel invocation
    int nthread = 256;
    int nblock = (N + nthread - 1)/nthread;
    VecAdd<<<nblock, nthread>>>(N, A, B, C);
    ...
}
```

In above example, the number of thread per block is set to 256, and `N` threads are dispatched into `[N/256]+1` blocks. In the last block, not all 256 threads are used. Therefore, the boundary check is necessary before a thread is applied.

The format of the device function is similar to a normal C function without the execution configuration declaration. Considering the execution efficiency in the kernel, the device function can be declared as an inline function to improve the performance by the declaration

__inline__.

Actually, the device function can also be called on the host side [24].

2.3 Summary

In this chapter, the architecture of the GPU and the abstraction of the CUDA are described. The 11th generation architecture of NVIDIA's GPU, Fermi, has 3.2 billion transistors to offer 512 cores grouped into 16 SMs making the code execution and data transfer parallelized. The many-core structure, high-throughput memory interface, and all new features including the 8× faster double precision floating point operation, the true cache hierarchy, the NVIDIA GigaThread™ Engine and ECC support, establish a substantial and consolidated foundation for the development of the massive-threading EMT simulation system, in which the data structure and parallel modules are designed based on these characteristics of GPU and CUDA.

3 Massive-threading Parallel EMT Simulator

The parallel EMT simulator [1] is designed by using the GPU based massive-threading parallel computing system introduced in chapters 1 and 2. The system framework, data interface and massive-threading parallel component and numerical method modules are the key components to realize the parallel EMT simulation system. Therefore, they are deliberately developed according to the features defined by the GPU architecture and CUDA abstraction to approach the maximum performance of the many-core parallel computing system.

3.1 Parallel Simulation System Framework

The system framework for the massive-threading parallel EMT simulation, including hardware and software, are based on the parallel features of GPU and CUDA. They are described in the section.

3.1.1 Hardware architecture of parallel simulation system

The hardware of the massive-threading parallel EMT simulation system is based on the heterogeneous cooperation between the CPU and the GPU. As shown in Fig. 3.1, the CPU is assembled on the center of mother board, beside which there are RAMs in the DDR3 slots, and the compute card C2050 with GPU and video memories inside is inserted in the PCIe slot.

The Table 3.1 lists the hardware specification of the parallel computing system. The

[1] This material has been submitted: Z. Zhou, and V. Dinavahi, "Parallel massive-thread electromagnetic transient simulation on GPU", *IEEE Trans. on Power Dilivery*, pp. 1-9, March 2012.

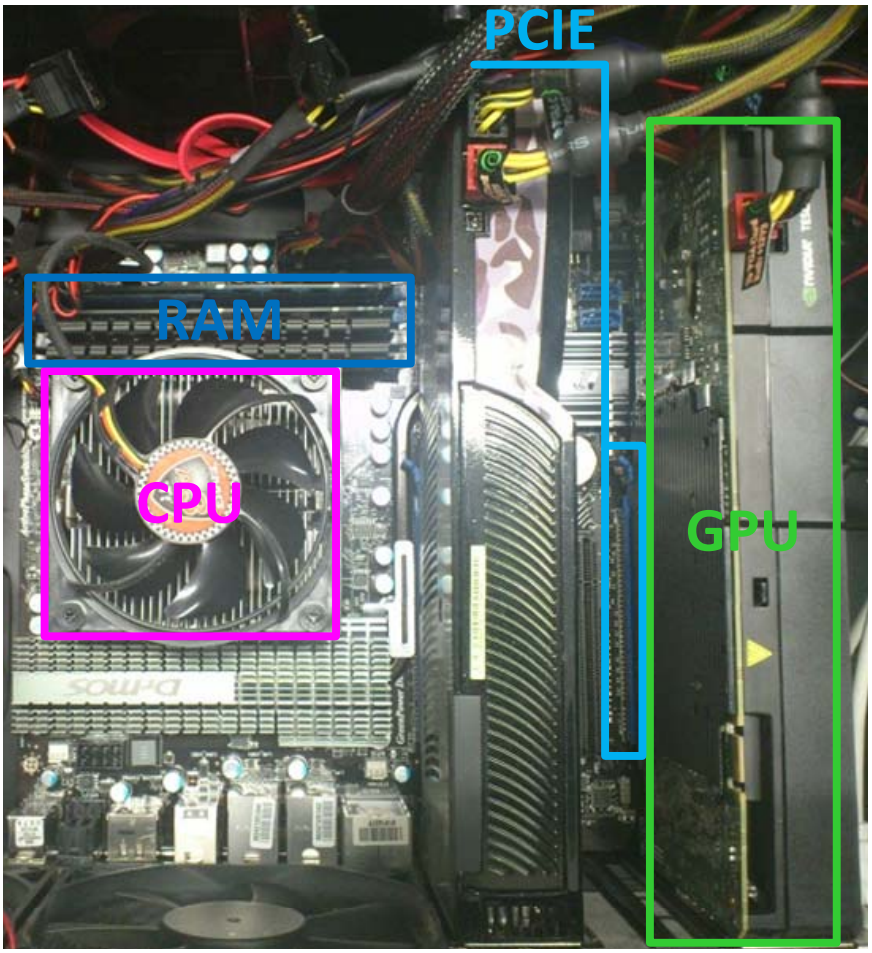

Figure 3.1: Physical layout of compute system depicting the CPU and GPU.

CPU has 4 cores running at 3.2GHz with 6MB L3 caches, which can drive the GPU efficiently. 16GB system RAM offers enough space for the data structure of large-scale power systems with thousands of buses, which will be compressed before transfer into the video memory of GPU since the data are sparse. The PCIe 2.0 interface offers 4GB/s bandwidth to link the CPU and GPU.

Table 3.1: Hardware Specification

CPU	AMD Phenom^{TM} II 955BE
Cores	4
Frequency	3.2 GHz
Cache	6 MB
System RAM	16 GB
GPU	C2050 (Fermi)
Device interface	PCIe 2.0

The specification of the parallel computing device is listed in Table 3.2. Two SMs of original GF100 chip are disabled for stable and thermal reasons; thus the C2050 has 14 SMs, 448 cores in total, running at 1.15GHz. There is a 3GB GDDR5 ECC supported memory running at 3GHz in the C2050 card.

3.1.2 Software architecture of parallel simulation system

The basic idea of EMT solution is to solve the linear equation created by applying nodal analysis to the power system circuit,

$$\boldsymbol{Y}\boldsymbol{v} = \boldsymbol{i}, \tag{3.1}$$

where $\boldsymbol{Y}$ is the admittance matrix, $\boldsymbol{v}$ is the vector of the nodal voltages and $\boldsymbol{i}$ is the vector of the currents injected into the relevant nodes [27]. Not all components in the power system are linear, some of them such as machines, surge arresters and power semiconductor devices are nonlinear. One solution is that all components are treated as nonlinear objects including the linear ones; however, the methods to solve the nonlinear problem, such as Newton-Raphson iteration method, are much more complicated and assume large computing load than linear solution. Another solution is to separate the computation into two parts, linear and nonlinear, and limit the iteration method only to the nonlinear components. Up to this time, the problem seems to be solved, however, a new problem is how to integrate linear and nonlinear solutions. Therefore, a interface, the compensation method interface, is utilized to connect the two parts, which will be expounded in the next section. Thus, the massive-threading parallel EMT simulator has two classes of modules for components. One is the linear modules, including the unique linear passive elements module (ULPEM), the universal line module (ULM), and the LU & forward-backward substitution module (LU-FBSM); the other is nonlinear modules, including the universal machine module (UMM) and the Newton-Raphson iteration module (NRIM).

In the CUDA architecture, the GPU works as the co-processor of the CPU. The CPU always starts the program, controls the process and collects the results. However, in order

Table 3.2: Parallel Computing Device Specification

Model	C2050
Number of SMs	14
Number of cores	448
Core clock	575 MHz
Shader clock	1.15 GHz
Memory	3 GB
Memory clock	3 GHz
TDP	225 W

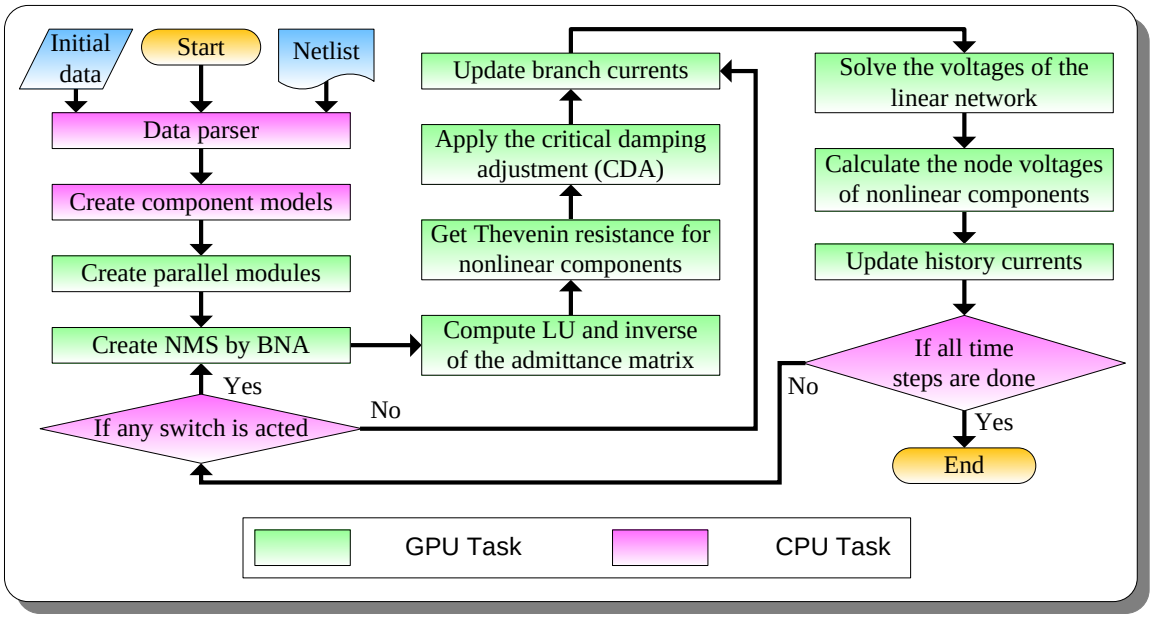

Figure 3.2: Flow chart of massive-threading parallel EMT simulation system.

to maximize the performance, as many tasks as possible tasks are assigned to the GPU. As shown in Fig. 3.2, the initial data and netlist are input to the host side when the simulation begins. All data are parsed and analyzed on the host side to create the component models, which are transferred to the device side. On the device side, the parallel modules are created according the the hardware CUDA parameter, from which the node mapping structure (NMS) of the linear system is created by the block node adjustment (BNA) method. The LU decomposition and the inverse of the admittance matrix, $\boldsymbol{Y}^{-1}$, are precalculated. Then the Thevenin equivalent resistance is extracted from $\boldsymbol{Y}^{-1}$. For the result stability of the power system simulation, the critical damping adjustment (CDA) [28] is applied when the system configuration is changed. After the branch currents, $\boldsymbol{i}$ on the right hand side of (3.1), are updated, the nodal voltages $\boldsymbol{v}$, on the left hand side of (3.1), are solved. Since the nonlinear components are interfaced to the power system with the compensation method module, their nodal voltages are calculated based on a superposition of the linear network solution with computed solution for the nonlinear component using Newton-Raphson iteration with the linear values input. With the final nodal voltages, the history currents for all components are updated. Thus, the calculation for one time-step is completed. If all time steps are finished, the whole simulation is done; otherwise, proceeds to the next time step. Before the simulation starts, the status of switches are checked. If any topology and configuration of the power system are changed, the NMS of the power system has to be rebuilt and continued with all following tasks; otherwise, the process goes to branch currents update directly. All procedures are repeated until all time steps of simulation are finished.

The CUDA specification of the parallel computing device used in this project, C2050, is

Table 3.3: CUDA system specification

CUDA runtime version	4.2
Compute capability	2.0
Multiprocessor	14
Maximum threads per multiprocessor	1536
Global memory	3 GB
Constant memory	64 KB
Shared memory per block	48 KB
Registers per block	32768
Maximum threads per block	1024
Maximum dimension of a block	$1024 \times 1024 \times 64$
Maximum dimension of a grid	$65535 \times 65535 \times 65535$

listed in Tab. 3.3, The compute capability of C2050 is 2.0. With compute capability 2.0 supported device in the 4.2 CUDA runtime library, the CUDA system has 3 GB global memory and 64 KB constant memory; and there are 48 KB shared memory and 32768 registers per block. The maximum thread dimensions in 3 directions are 1024, 1024 and 64 for one block, in which there are a maximum of 1024 threads. The maximum block dimensions in 3 directions are 65535, 65535 and 65535 for one grid, in which there is a maximum of 1536×14 threads, since there are a maximum of 1536 threads per multiprocessor and there are 14 multiprocessors.

3.2 Compensation Method Interface

The compensation method [29] is applied in EMT simulation to partition the linear and nonlinear calculation into relatively independent sections instead of solving the entire nonlinear network simultaneously. In practice, it can effectively increase the speed and reliability of EMT simulation because of the smaller partitioned nonlinear system scale. Although the compensation method is not perfect, limitations of compensation method, such as only one nonlinear component is applicable connecting to one node, can be solved by several ways, such as to add artificial delays between the components, in which the error and deviation are controlled within a reasonable margin. Therefore, the compensation method is widely used in EMT simulation to reduce the amount of computation due to its compromise between implementation and accuracy.

As shown in Fig. 3.3 (a), the nonlinear component, connecting to node $\boldsymbol{m}$ and $\boldsymbol{k}$, is isolated from the n-node power system. On the linear network side, the compensation equations are given as

$$\boldsymbol{v_{km}} = \boldsymbol{v_{kmo}} - \boldsymbol{R}_{eq}\boldsymbol{i_{km}}, \tag{3.2}$$

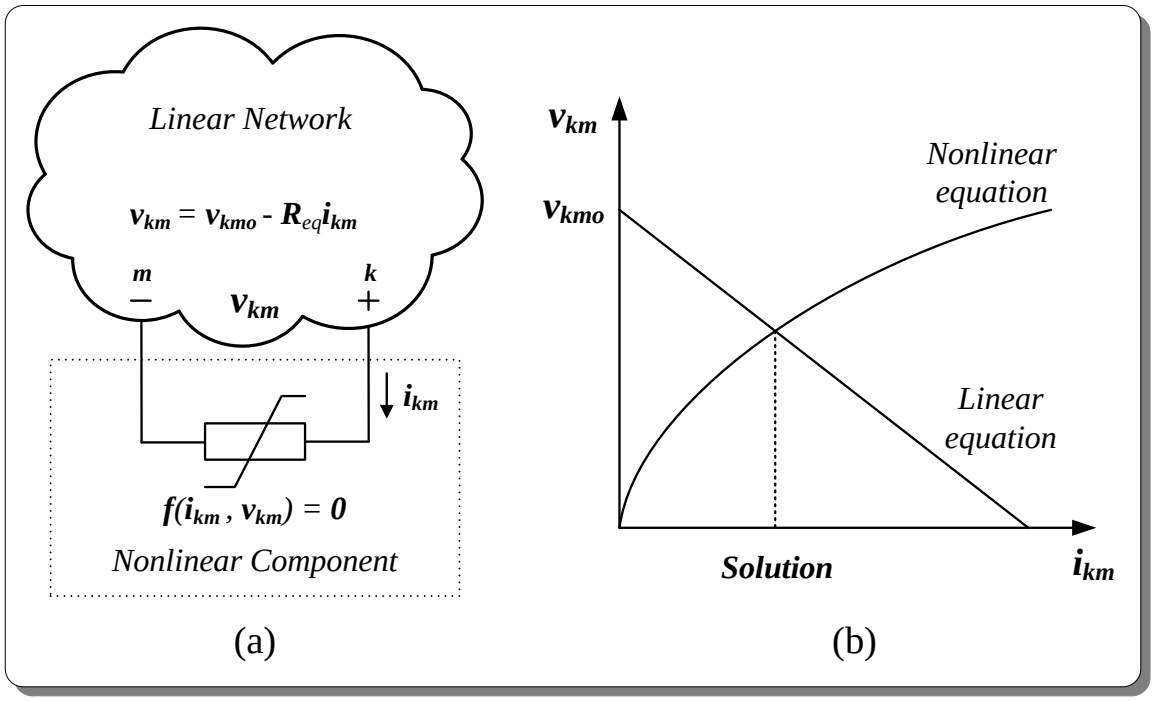

Figure 3.3: Compensation method interface. (a) Power system with nonlinear component, and (b) Nonlinear solution.

where $\boldsymbol{v_{km}}$ is a $p \times 1$ vector of voltages across p-phase nonlinear component, $\boldsymbol{i_{km}}$ is a $p \times 1$ vector of currents through p-phase nonlinear component, $\boldsymbol{v_{kmo}}$ is a $p \times 1$ vector of open-circuit voltages (without the nonlinear branches) between nodes $\boldsymbol{m}$ and $\boldsymbol{k}$, and $\boldsymbol{R}_{eq}$ is a $p \times p$ Thevenin equivalent resistance matrix of the linear network. As shown in Fig. 3.4, $\boldsymbol{r}_{th}^{\boldsymbol{k}}$ and $\boldsymbol{r}_{th}^{\boldsymbol{m}}$ are $n \times p$ matrices, where n is the size of the inverse admittance matrix $\boldsymbol{Y}^{-1}$ and p is the number of phases, defining the Thevenin resistance of $\boldsymbol{k}$ and $\boldsymbol{m}$ nodes to ground, which are the $\boldsymbol{k}$ and $\boldsymbol{m}$ columns of $\boldsymbol{Y}^{-1}$, where $\boldsymbol{k}$ and $\boldsymbol{m}$ have p phases. Then, the Thevenin resistances from $\boldsymbol{k}$ to $\boldsymbol{m}$, $\boldsymbol{r}_{th}^{\boldsymbol{km}}$, an $n \times p$ matrix, are given by the difference of $\boldsymbol{r}_{th}^{\boldsymbol{k}}$ and $\boldsymbol{r}_{th}^{\boldsymbol{m}}$. Thus, the equivalent resistance matrix $\boldsymbol{R}_{eq}$ in 3.2 is calculated by

$$\boldsymbol{R}_{eq}[i][j] = \boldsymbol{r}_{th}^{\boldsymbol{km}}[\boldsymbol{k}[i]][j] - \boldsymbol{r}_{th}^{\boldsymbol{km}}[\boldsymbol{m}[i]][j] \qquad (i, j = \{1, 2, \cdots, p\}). \tag{3.3}$$

On the nonlinear component side, the current and voltage characteristics are expressed as

$$\boldsymbol{f}(\boldsymbol{i_{km}}, \boldsymbol{v_{km}}) = \boldsymbol{0}, \tag{3.4}$$

where $\boldsymbol{f}$ is the nonlinear functions. When the linear equations 3.2 and nonlinear equations 3.4 are solved shown in Fig. 3.3 (b), the solutions of the currents $\boldsymbol{i_{km}}$ can be obtained. With the superimposition $\boldsymbol{i_{km}}$ on the linear network solution as current sources, the solution of the entire network is calculated as

$$\boldsymbol{v} = \boldsymbol{v_o} - \boldsymbol{r}_{th}^{\boldsymbol{km}} \boldsymbol{i_{km}}, \tag{3.5}$$

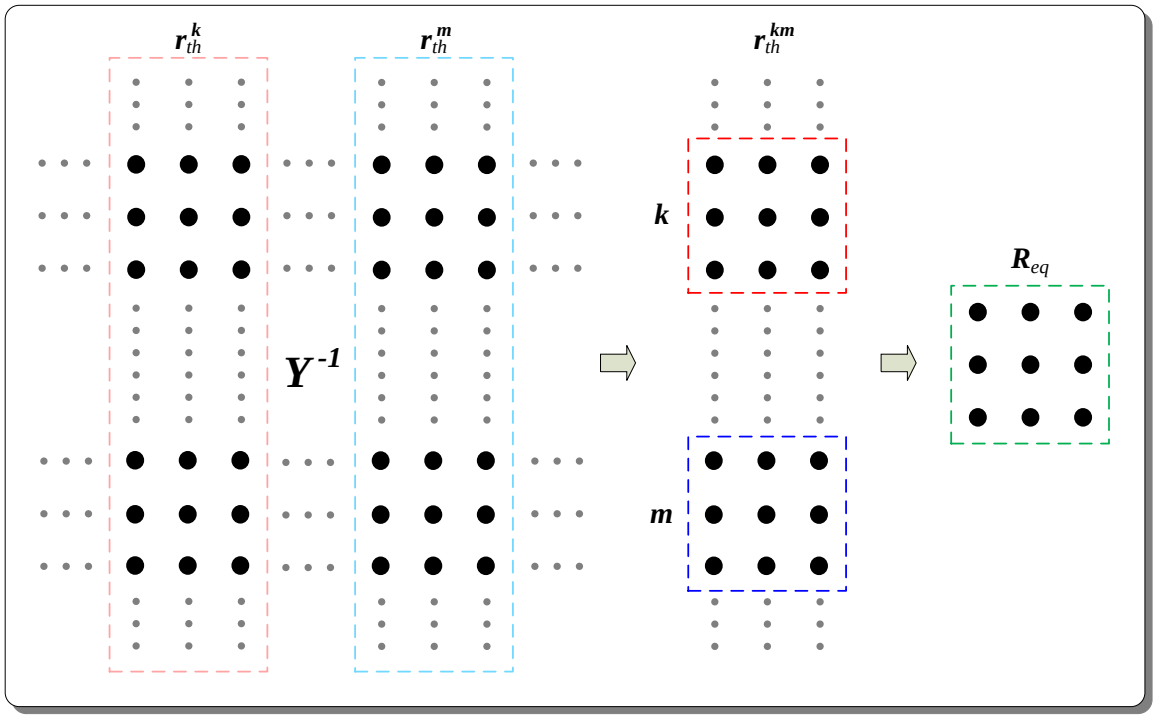

Figure 3.4: $\boldsymbol{R}_{eq}$ calculation for the compensation methods interface (CMI).

where $\boldsymbol{v}$ is a $n \times 1$ vector of entire network solutions and $\boldsymbol{v_o}$ is $n \times 1$ vector of open circuit linear network solutions (without nonlinear components).

In the parallel computing system, all nonlinear components with compensation method interface can be calculated simultaneously instead of solving them one by one since all nonlinear components are decoupled. Moreover, because the compensation method constrain the computing dimension of nonlinear solution, which uses iteration method normally, it reduces the amount of computation effectively.

3.3 Massive-Threading Parallel Component and Method Modules

The three main classes of components considered in this EMT simulation are the linear passive elements, transmission line and synchronous machines, and two major solver methods for nonlinear and linear system that are the Newton-Raphson iteration and forward-backward substitution with LU factorization methods implemented in parallel. In order to build up a flexible and extendable EMT simulation system, all these models and methods are modularized into independent modules. As shown in Fig. 3.5, after creating the parallel data structure by netlist and initial data, all component modules, including unique linear passive element module (ULPEM), universal line module (ULM) and universal machine module (UMM), accomplish the computations using the solver modules, including Newton-Raphson iteration module (NRIM) and LU & forward-backward substitution

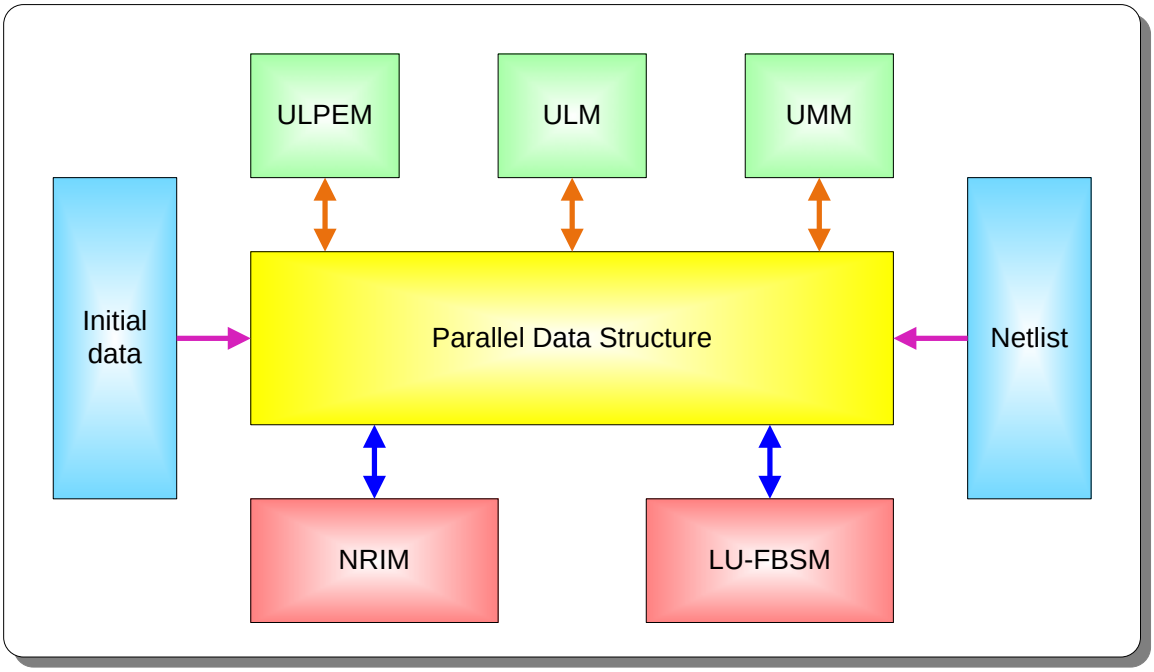

Figure 3.5: Parallel EMT simulation system diagram.

module (LU-FBSM). Via this modularized system architecture, the EMT simulator can be easily extended, upgraded and maintained with new component models and numerical methods in the future.

3.3.1 Linear passive elements (LPE)

3.3.1.1 Model formulation

Linear passive elements (LPEs), such as resistance, inductance, capacitance, switches and their combinations, are represented by a discrete-time lumped model [30]. As mentioned in Section II, since all threads in a kernel run the same instruction concurrently, a unified model is required for all LPEs in the system. Using the Trapezoidal rule of integration, any LPE combination can be modeled as a discrete Norton equivalent circuit comprising of an equivalent conductance and a history current source. In the unified model, every LPE has a R, L or C character. An arbitrary LPE Z shown in Fig. 3.6 (a) is a combination of L, R and C shown in Fig. 3.6 (b). The voltage relations are given as

$$v(t) = v_L(t) + v_R(t) + v_C(t), \tag{3.6}$$

where $v(t)$ is the voltage of the unified LPE Z, $v_L(t)$, $v_R(t)$ and $v_C(t)$ are the voltages of the inductor, resistor and capacitor respectively. The current and voltage relations of them are

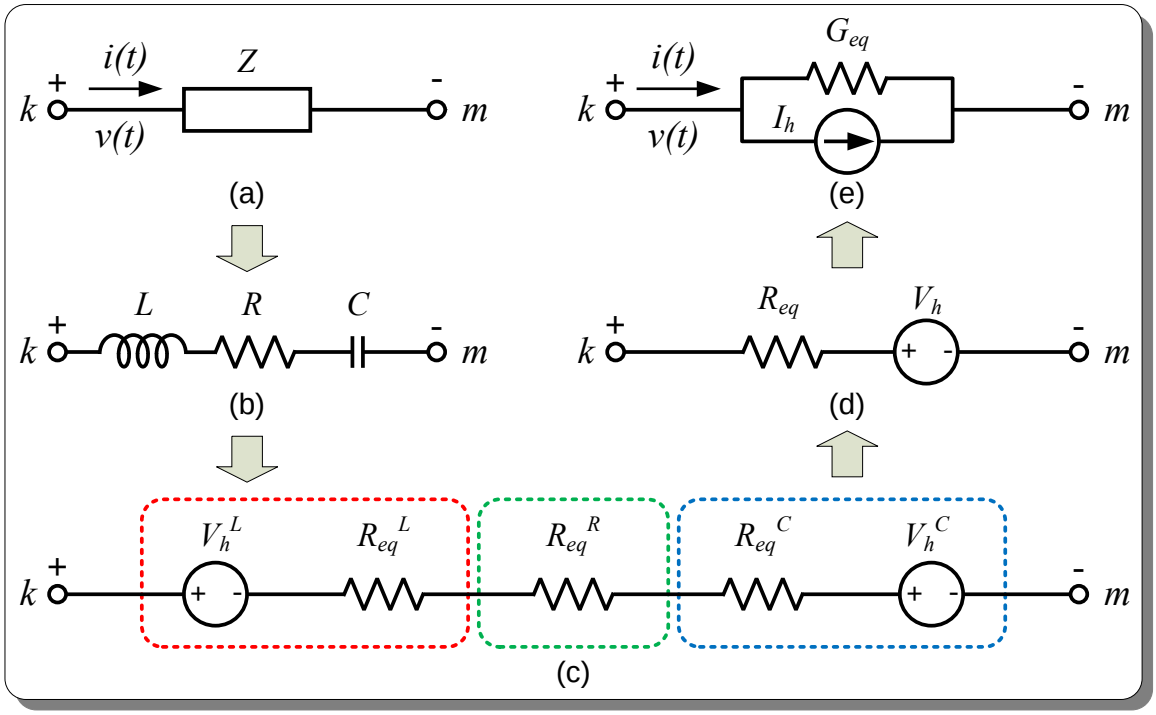

Figure 3.6: Unified linear passive element lumped module.

expressed as

$$v_L(t) = L\frac{di(t)}{dt}, \tag{3.7}$$

$$v_R(t) = Ri(t), \tag{3.8}$$

$$v_C(t) = \frac{1}{C}\int i(t)dt, \tag{3.9}$$

where $i_L(t)$, $i_R(t)$ and $i_C(t)$ are the currents of the inductor, resistor and capacitor respectively. Applying the Trapezoidal rule for L and C gives the discretized equations as

$$R_{eq}^L i(t) = v_L(t) + V_h^L(t - \Delta t), \tag{3.10}$$

$$R_{eq}^C i(t) = v_C(t) + V_h^C(t - \Delta t), \tag{3.11}$$

where the equivalent resistors R_{eq}^L and R_{eq}^C for inductor and capacitor are defined as

$$R_{eq}^L = \frac{2L}{\Delta t}, \tag{3.12}$$

$$R_{eq}^C = \frac{\Delta t}{2C}. \tag{3.13}$$

The history voltages $V_h^L(t - \Delta t)$ and $V_h^C(t - \Delta t)$ for inductor and capacitor in (3.10) and (3.11) are calculated with following recurrence equations

$$V_h^L(t - \Delta t) = -V_h^L(t - 2\Delta t) + 2R_{eq}^L i(t - \Delta t), \tag{3.14}$$

$$V_h^C(t - \Delta t) = \quad V_h^C(t - 2\Delta t) - 2R_{eq}^C i(t - \Delta t). \tag{3.15}$$

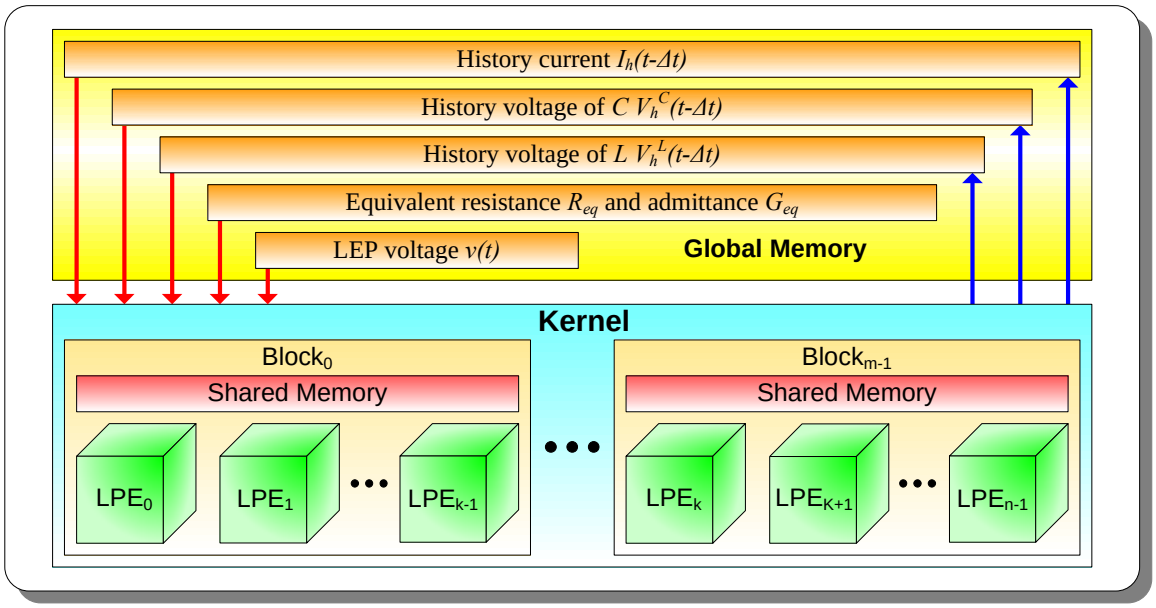

Figure 3.7: Massive-threading parallel ULPEM.

Fig. 3.6 (c) shows the Thevenin equivalent circuit after discretization, which includes 3 equivalent resistors and 2 history voltage sources. They can be regrouped by (3.6) as shown in Fig. 3.6(d), where the integrated equivalent resistor and history voltage source of the LPE Z is expressed as

$$R_{eq} = R_{eq}^R + R_{eq}^C + R_{eq}^L, \tag{3.16}$$

$$V_h(t-\Delta t) = V_h^L(t-\Delta t) + V_h^C(t-\Delta t). \tag{3.17}$$

Source transformation results in the Norton equivalent circuit shown in Fig. 3.6(e), whose parameters are given as

$$G_{eq} = 1/R_{eq}, \tag{3.18}$$

$$I_h(t-\Delta t) = G_{eq}V_h(t-\Delta t), \tag{3.19}$$

where G_{eq} is the equivalent conductance and I_h is the history current source of the unified model. The LPE current $i(t)$ is updated as

$$i(t) = G_{eq}v(t) + I_h(t-\Delta t). \tag{3.20}$$

With the unified LPE model (ULPEM), all linear elements can be processed in the same kernel.

3.3.1.2 Massive-thread parallel implementation

As shown in Fig. 3.7, the designed parallel module for unified LPE only has one kernel for the computation. For each LPE, a CUDA thread is assigned to execute the computation

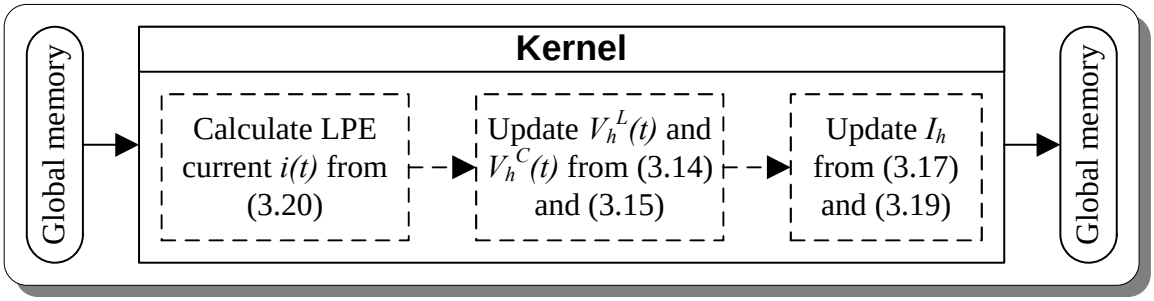

Figure 3.8: Kernel operation flow in the ULPEM.

based on the SIMT format. When the number n of LPEs exceeds the limitations of thread per block k, they will be divided into m groups assigned to multiple CUDA blocks:

$$m = \left[\frac{n-1}{k}\right] + 1, \tag{3.21}$$

where n, k and m are integers.

The operation flow of the LPE kernel is shown in Fig. 3.8. Inside the kernel, the LPE current $i(t)$ is computed from (3.20) firstly, then the inductive and capacitive history voltages, $V_h^L(t-\Delta t)$ and $V_h^C(t-\Delta t)$, are updated from (3.14) and (3.15) using the values of last step, and finally the history current $I_h(t-\Delta t)$ is updated using (3.17) and (3.19). The equivalent resistance R_{eq} and equivalent admittance G_{eq} are reused for all time steps unless the network configuration is changed by switches. The only global memory accesses are reading the input variables and writing the output variables, and all computations of LPE take place inside the threads. During the EMT simulation, all variables are stored and reused on the device side; thus the host-device and device-host data transmission is minimized in each time step.

3.3.2 Transmission lines

3.3.2.1 Model formulation

The universal line model (ULM) is a phase-domain wide-band fully frequency-dependent line model [31] capable of representing both symmetrical and asymmetrical overhead transmission lines and underground cables. Traditional frequency dependent transmission line models [32] were constituted in the modal-domain based on constant transformative matrices with frequency-dependent model for the traveling waves; therefore, the applicability of these models was restricted to symmetrical (transposed) lines and cables. The ULM avoid the transformation matrices and is constituted directly in the phase domain; however it involves computationally expensive convolutions.

3.3.2.1.1 Frequency-domain formulation The solution of the traveling wave equations at the sending-end 'k' and the receiving-end 'm' of a p-phase, l-length transmission line,

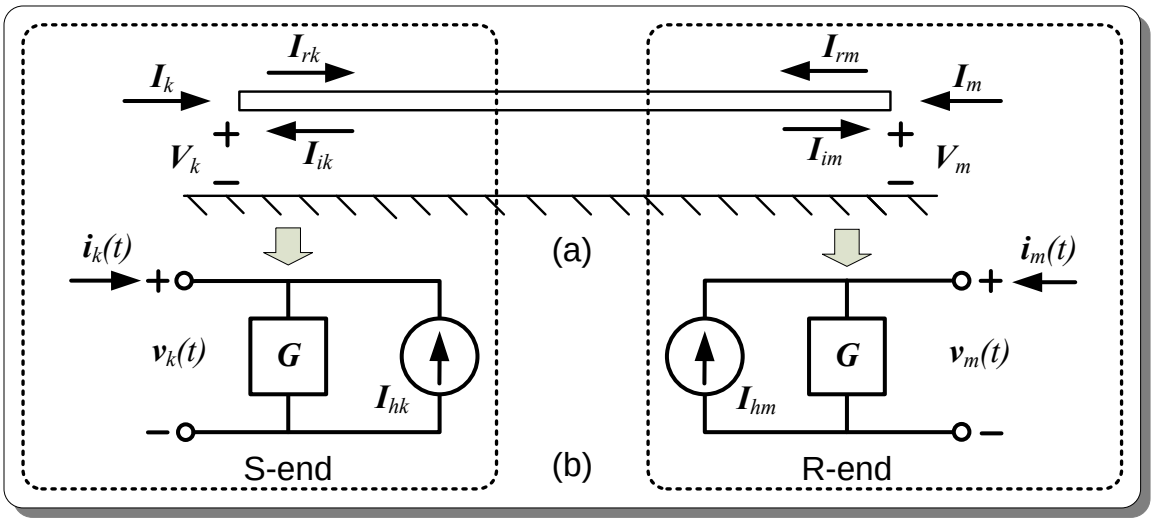

Figure 3.9: Universal line model.

shown in Fig. 3.9 (a), are expressed as.

$$\boldsymbol{I}_k = \boldsymbol{Y}\boldsymbol{V}_k - 2\boldsymbol{I}_{ik}, \tag{3.22a}$$

$$\boldsymbol{I}_m = \boldsymbol{Y}\boldsymbol{V}_m - 2\boldsymbol{I}_{im}, \tag{3.22b}$$

where $\boldsymbol{i}_i$ are the incident currents expressed at the two ends as

$$\boldsymbol{I}_{ik} = \boldsymbol{H}\boldsymbol{I}_{rm}, \tag{3.23a}$$

$$\boldsymbol{I}_{im} = \boldsymbol{H}\boldsymbol{I}_{rk}. \tag{3.23b}$$

$\boldsymbol{Y}$ and $\boldsymbol{H}$ in (3.22) and (3.23) are the characteristic admittance and the propagation matrices, expressed as

$$\boldsymbol{Y} = \sqrt{\frac{\boldsymbol{y}}{\boldsymbol{z}}}, \tag{3.24a}$$

$$\boldsymbol{H} = e^{-\sqrt{\boldsymbol{y}\boldsymbol{z}}l}, \tag{3.24b}$$

where $\boldsymbol{y}$ and $\boldsymbol{z}$ are shunt admittance and series impedance matrices per unit length. For the time-domain implementation, they are approximated by finite-order rational functions using the vector fitting (VF) method [33]. The admittance matrix $\boldsymbol{Y}$ in (3.24a) can be fitted directly in the phase domain. An element of $\boldsymbol{Y}$ is expressed as

$$\boldsymbol{Y}_{(i,j)}(s) = \sum_{m=1}^{N_p} \frac{\boldsymbol{r}_{Y(i,j)}(m)}{s - \boldsymbol{p}_Y(m)} + \boldsymbol{d}(i,j), \tag{3.25}$$

where $\boldsymbol{r}_Y$, $\boldsymbol{p}_Y$, $\boldsymbol{d}$ and N_p are the residues, poles, proportional terms, and the number of poles of $\boldsymbol{Y}$ respectively. Thus, all elements of $\boldsymbol{Y}$ have identical poles $\boldsymbol{p}_Y$. The fitting of $\boldsymbol{H}$ in (3.24b) is slightly different, because it has various modes. Before it is fitted in the phase

domain, $\boldsymbol{H}$ has to be fitted in each mode with poles and time delays. The general element of $\boldsymbol{H}$ is expressed as

$$\boldsymbol{H}_{(i,j)}(s) = \sum_{k=1}^{N_g} \left[\sum_{n=1}^{\boldsymbol{N}_p(k)} \frac{\boldsymbol{r}_{H_{(i,j)}}^{(k)}(n)}{s - \boldsymbol{p}_H{}^{(k)}(n)} \right] e^{-s\boldsymbol{\tau}(k)}, \tag{3.26}$$

where N_g is the number of modes; $\boldsymbol{N}_p(k)$ and $\boldsymbol{\tau}(k)$ are numbers of poles and time delays for fitting the k^{th} mode; and $\boldsymbol{r}_H{}^{(k)}$ and $\boldsymbol{p}_H{}^{(k)}$ are residues and poles for the k^{th} mode. Again, the poles are identical for all elements in each mode.

3.3.2.1.2 Time-domain implementation Modeled as two decoupled Norton equivalent circuits, shown in Fig. 3.9 (b), the current and voltage relation at both ends are given as

$$\boldsymbol{i}_k(t) = \boldsymbol{G}_Y \boldsymbol{v}_k(t) - \boldsymbol{I}_{hk}, \tag{3.27a}$$

$$\boldsymbol{i}_m(t) = \boldsymbol{G}_Y \boldsymbol{v}_m(t) - \boldsymbol{I}_{hm}, \tag{3.27b}$$

where the history currents $\boldsymbol{I}_h$ at the two ends of the line are expressed as

$$\boldsymbol{I}_{hk} = \boldsymbol{Y} * \boldsymbol{v}_k(t) - 2\boldsymbol{H} * \boldsymbol{i}_{rm}(t-\tau), \tag{3.28a}$$

$$\boldsymbol{I}_{hm} = \boldsymbol{Y} * \boldsymbol{v}_m(t) - 2\boldsymbol{H} * \boldsymbol{i}_{rk}(t-\tau), \tag{3.28b}$$

where the "$*$" denotes numerical complex matrix-vector convolution since the poles $\boldsymbol{p}_Y$ and $\boldsymbol{p}_H$ can be complex numbers. The equivalent conductance matrix $\boldsymbol{G}_Y$ in (3.27) is given as

$$\boldsymbol{G}_Y = \boldsymbol{d} + \boldsymbol{r}_Y \boldsymbol{\lambda}_Y, \tag{3.29}$$

where the coefficients $\boldsymbol{\lambda}_Y$ are defined as

$$\boldsymbol{\lambda}_Y = (\frac{\Delta t}{2})/(1 - \boldsymbol{p}_Y \frac{\Delta t}{2}), \tag{3.30}$$

with Δt being the simulation time step. The numerical convolution $\boldsymbol{Y} * \boldsymbol{v}(t)$ in (3.28) is defined as

$$\boldsymbol{Y} * \boldsymbol{v}(t) = \boldsymbol{c}_Y \boldsymbol{x}_Y(t), \tag{3.31}$$

where the coefficients $\boldsymbol{c}_Y$ are given as

$$\boldsymbol{c}_Y = \boldsymbol{r}_Y(\boldsymbol{\alpha}_Y + 1)\boldsymbol{\lambda}_Y, \tag{3.32}$$

and the state variables $\boldsymbol{x}_Y$ are defined as

$$\boldsymbol{x}_Y(t) = \boldsymbol{\alpha}_Y \boldsymbol{x}_Y(t - \Delta t) + \boldsymbol{v}(t - \Delta t) \tag{3.33}$$

with the coefficients $\boldsymbol{\alpha}_Y$ expressed as

$$\boldsymbol{\alpha}_Y = (1 + \boldsymbol{p}_Y \frac{\Delta t}{2})/(1 - \boldsymbol{p}_Y \frac{\Delta t}{2}). \tag{3.34}$$

Similarly, the numerical convolution $\boldsymbol{H} * \boldsymbol{i}_r(t-\tau)$ in (3.28) is defined as

$$\boldsymbol{H} * \boldsymbol{i}_r(t-\tau) = \boldsymbol{c}_H \boldsymbol{x}_H(t) + \boldsymbol{G}_H \boldsymbol{i}_r(t-\tau), \tag{3.35}$$

where the coefficients $\boldsymbol{c}_H$ are given as

$$\boldsymbol{c}_H = \boldsymbol{r}_H(\boldsymbol{\alpha}_H + 1)\boldsymbol{\lambda}_H; \tag{3.36}$$

and the state variables $\boldsymbol{x}_H(t)$ are defined as

$$\boldsymbol{x}_H(t) = \boldsymbol{\alpha}_H \boldsymbol{x}_H(t-\Delta t) + \boldsymbol{i}_r(t-\tau-\Delta t), \tag{3.37}$$

with the coefficients $\boldsymbol{\alpha}_H$ expressed as

$$\boldsymbol{\alpha}_H = (1 + \boldsymbol{p}_H \frac{\Delta t}{2})/(1 - \boldsymbol{p}_H \frac{\Delta t}{2}). \tag{3.38}$$

The propagation matrix $\boldsymbol{G}_H$ in (3.35) is given as

$$\boldsymbol{G}_H = \sum_{1}^{N_g} \boldsymbol{r}_Y \boldsymbol{\lambda}_Y. \tag{3.39}$$

The reflected current $\boldsymbol{i}_r$ above are defined at the two ends as

$$\boldsymbol{i}_{rk}(t) = \boldsymbol{i}_k(t) - \boldsymbol{i}_{ik}(t), \tag{3.40a}$$

$$\boldsymbol{i}_{rm}(t) = \boldsymbol{i}_m(t) - \boldsymbol{i}_{im}(t), \tag{3.40b}$$

where the ULM currents $\boldsymbol{i}(t)$ are given by (3.27) and the incident currents $\boldsymbol{i}_i(t)$ are defined at the two ends as

$$\boldsymbol{i}_{ik}(t) = \boldsymbol{H} * \boldsymbol{i}_{rk}(t-\tau), \tag{3.41a}$$

$$\boldsymbol{i}_{im}(t) = \boldsymbol{H} * \boldsymbol{i}_{rm}(t-\tau), \tag{3.41b}$$

where the convolution is given by (3.35), and $\boldsymbol{i}_r(t-\tau)$ are the reflected currents before time delay τ.

3.3.2.1.3 Interpolation Since the wave traveling time τ is not an integral multiple of the time step Δt normally, linear interpolation is used to approximate the reflected current $\boldsymbol{i}_r(t-\tau)$ in (3.35) and $\boldsymbol{i}_r(t-\tau-\Delta t)$ in (3.37). The time delay τ can be expressed as

$$\tau = (N + \delta)\Delta t, \qquad (0 \leqslant \delta < 1) \tag{3.42}$$

where N is an integer and δ is a real number. As shown in Fig. 3.10, since the time-step Δt is small enough, the curves between known points $i_r(t-(N+2)\Delta t)$, $i_r(t-(N+1)\Delta t)$ and $i_r(t-N\Delta t)$ are replaced by lines. Thus, the unknown current $i_r(t-\tau-\Delta t)$ and $i_r(t-\tau)$ are approximated by the linear interpolation as

$$i_r(t-\tau-\Delta t) = (1-\delta) i_r(t-(N+1)\Delta t) + \delta i_r(t-(N+2)\Delta t), \tag{3.43a}$$

$$i_r(t-\tau) = (1-\delta) i_r(t-N\Delta t) + \delta i_r(t-(N+1)\Delta t). \tag{3.43b}$$

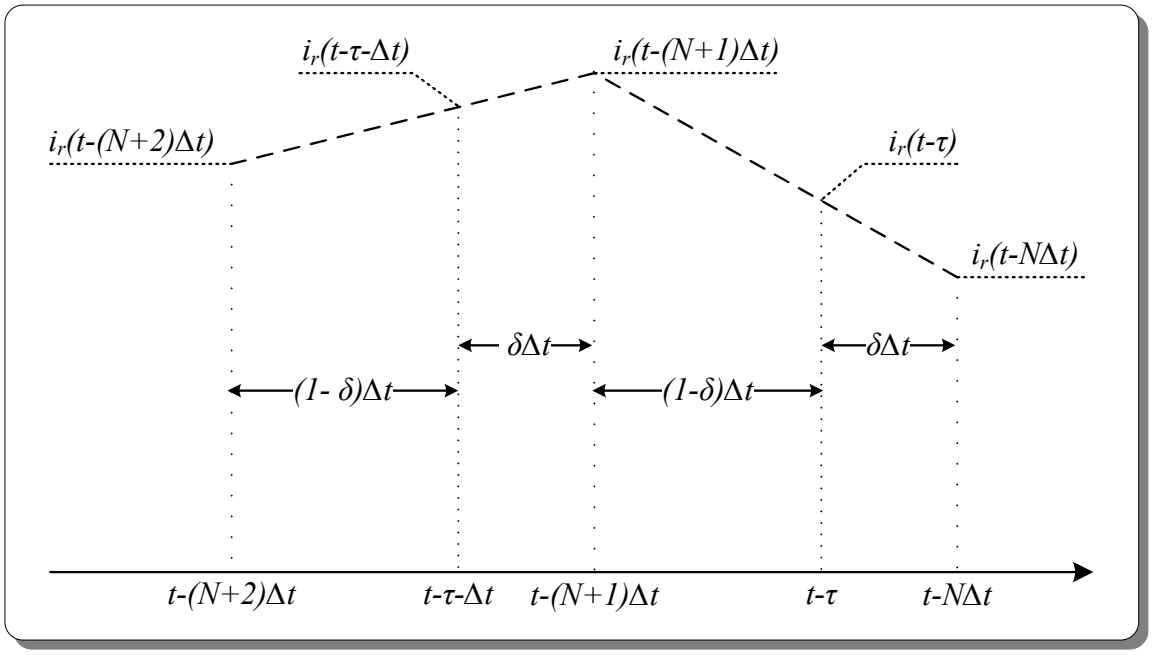

Figure 3.10: Linear interpolation.

3.3.2.2 Massive-thread parallel implementation

As shown in Fig. 3.11, the designed parallel module for ULM includes 8 kernels grouped into 4 stages. Stage 1 updates the reflected currents $\boldsymbol{i}_r$ and calculates the interpolation for the reflected currents before delay $\boldsymbol{\tau}$; Stage 2 updates the state variable $\boldsymbol{x}(t)$; Stage 3 computes the convolutions; and Stage 4 updates the incident current $\boldsymbol{i}_i$ the history current $\boldsymbol{I}_h$. All the kernels inside the same stage are executed concurrently in the Fermi architecture space.The computation for each ULM unit is done by a CUDA block running in SIMT, inside which multiple threads are assigned to handle vector and matrix operations based on SIMD. Therefore, every kernel has n blocks (the number of ULM units) in every stage, and the number of threads in a block depends on the dimension of computed vectors and matrices, which is typically based on the number of poles and residues from vector fitting. The data are transferred deliberately from the global memory into the shared memory first to improve the memory access performance due to the critical bandwidth requirement of vector and matrix operations. In order to calculate the reflected currents $i_r(t-\tau)$, a FIFO deep in $N+2$ is designed to store the history values of $i_r(t)$. For instance, a 15-value FIFO is needed when $\tau = 13.35\Delta t$ since N is 13 in this case.

The kernel operation flow of the ULM module is shown in Fig. 3.12. Since the thread dimension and shared memory size have to be reconfigured in different tasks, such as in updating variables, interpolation and convolutions, they are separated into different kernels, and their results are output to global memory and shared with other kernels. $\boldsymbol{i}_r(t)$ are updated from (3.40) in **Kernel$_0$** and $\boldsymbol{i}_r(t-\tau)$ are calculated from (3.43) by interpolation

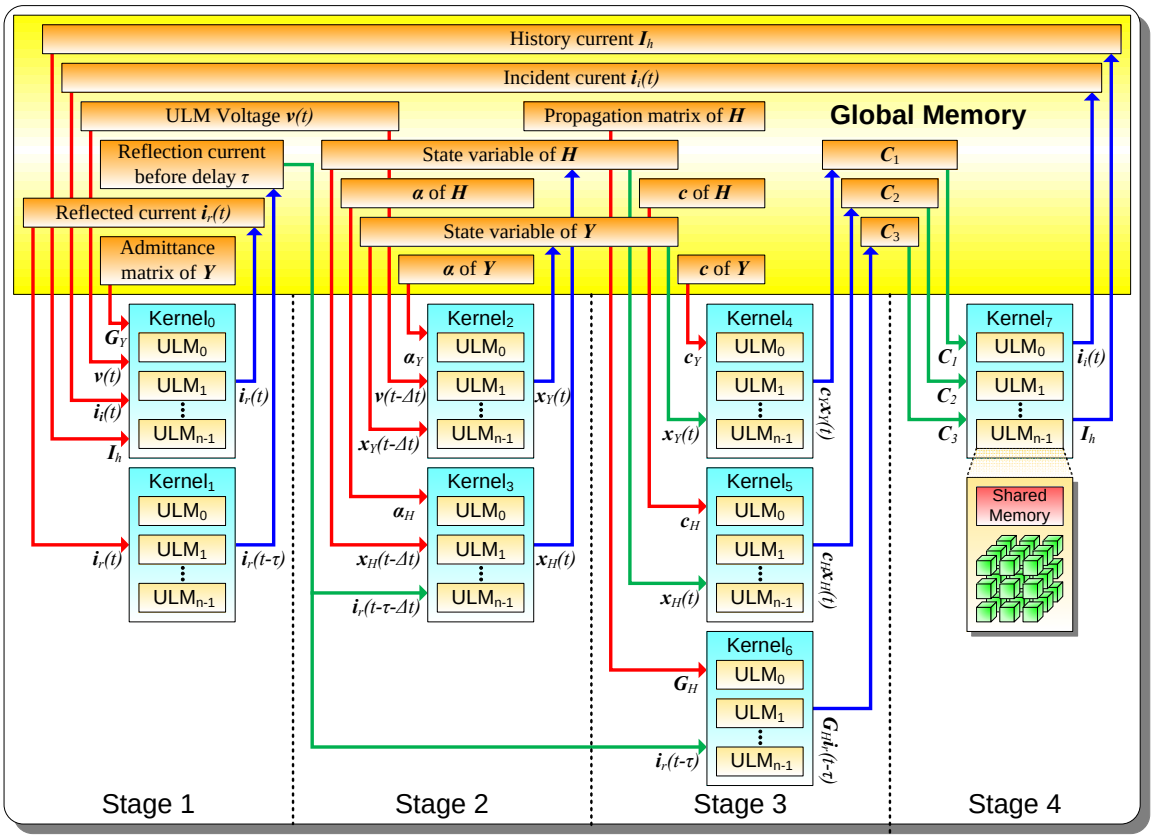

Figure 3.11: Massive-threading parallel ULM.

with previous values in **Kernel$_1$**. The state variables $\boldsymbol{x}_Y(t)$ with the coefficients $\boldsymbol{\alpha}_H$ and $\boldsymbol{\alpha}_Y$, which are precalculated and stored in the global memory, in **Kernel$_2$** and **Kernel$_3$** respectively. All convolutions $\boldsymbol{c}_Y\boldsymbol{x}_Y(t)$, $\boldsymbol{c}_H\boldsymbol{x}_H(t)$ and $\boldsymbol{G}_H\boldsymbol{i}_r(t-\tau)$ are computed from (3.31) and (3.35) in **Kernel$_4$**, **Kernel$_5$** and **Kernel$_6$** concurrently. In **Kernel$_7$**, $\boldsymbol{i}_i$ are updated first from (3.41), and then the history current $\boldsymbol{I}_h$ are updated from (3.28) finally. Since the parallel computation is based on each ULM unit instead of its sending and receiving ends, all the variables of 'k' and 'm' are computed within one kernel, avoiding the data exchange between 'k' and 'm' ends. Similar to the LPE module, all the module variables are limited to the device side, i.e. to the global and shared memories of the GPU; thus there is no data exchange between host and device during ULM execution.

3.3.3 Electrical Machines

3.3.3.1 Model formulation

There are several types of rotating machine models that can be used for EMT studies. The advantage of the unified machine model (UMM) [34] [35] is that it provides a unified mathematical framework to model up to 12 types of rotating machines including asynchronous, synchronous and DC machines. The electrical part of the UMM includes the armature and

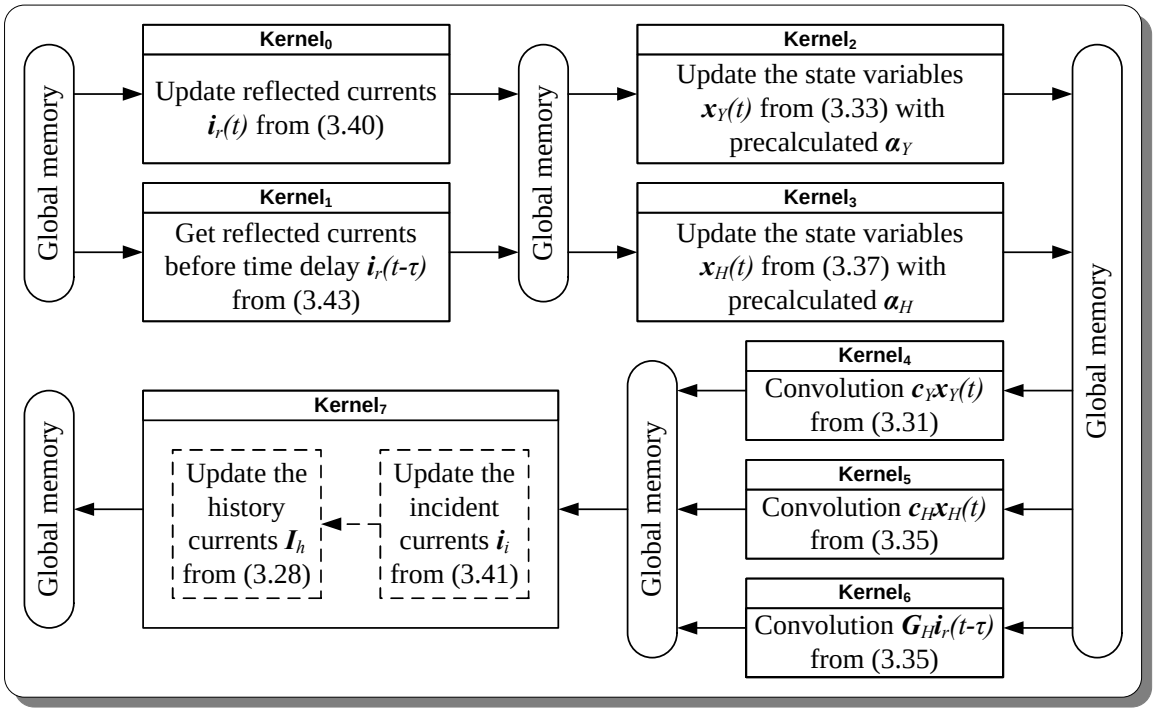

Figure 3.12: Kernel operation flow in the ULM.

field windings. The UMM is allowed to have up to 3 armature windings (converted to 3 *dq0* windings), and an unlimited number of windings on the field structure. The mechanical part of the UMM is modeled as an equivalent lumped electric network, where the electromagnetic torque appears as current source. An alternate representation of the mechanical part as a multi-mass model (up to a maximum of 6 masses representing various turbine stages) is also possible.

3.3.3.1.1 Electrical Part In the UMM used in this project, without loss of generality, there are 3-phase stator armature windings $\{a, b, c\}$, one field winding f, up to 2 damper windings $\{D_1, D_2\}$ on the rotor direct d-axis, and up to 3 damper windings $\{Q_1, Q_2, Q_3\}$ on the rotor quadrature q-axis, as shown in Fig. 3.13. Thus there are a maximum of 9 coupled windings whose discretized winding equations are described as

$$\boldsymbol{v}_{dq0}(t) = -\boldsymbol{R}\boldsymbol{i}_{dq0}(t) - \frac{2}{\Delta t}\boldsymbol{\lambda}_{dq0}(t) + \boldsymbol{u}(t) + \boldsymbol{V}_h, \tag{3.44}$$

where $\boldsymbol{R}$ is the winding resistance matrix, and the flux linkage $\boldsymbol{\lambda}_{dq0}$ is given as

$$\boldsymbol{\lambda}_{dq0}(t) = \boldsymbol{L}\boldsymbol{i}_{dq0}(t), \tag{3.45}$$

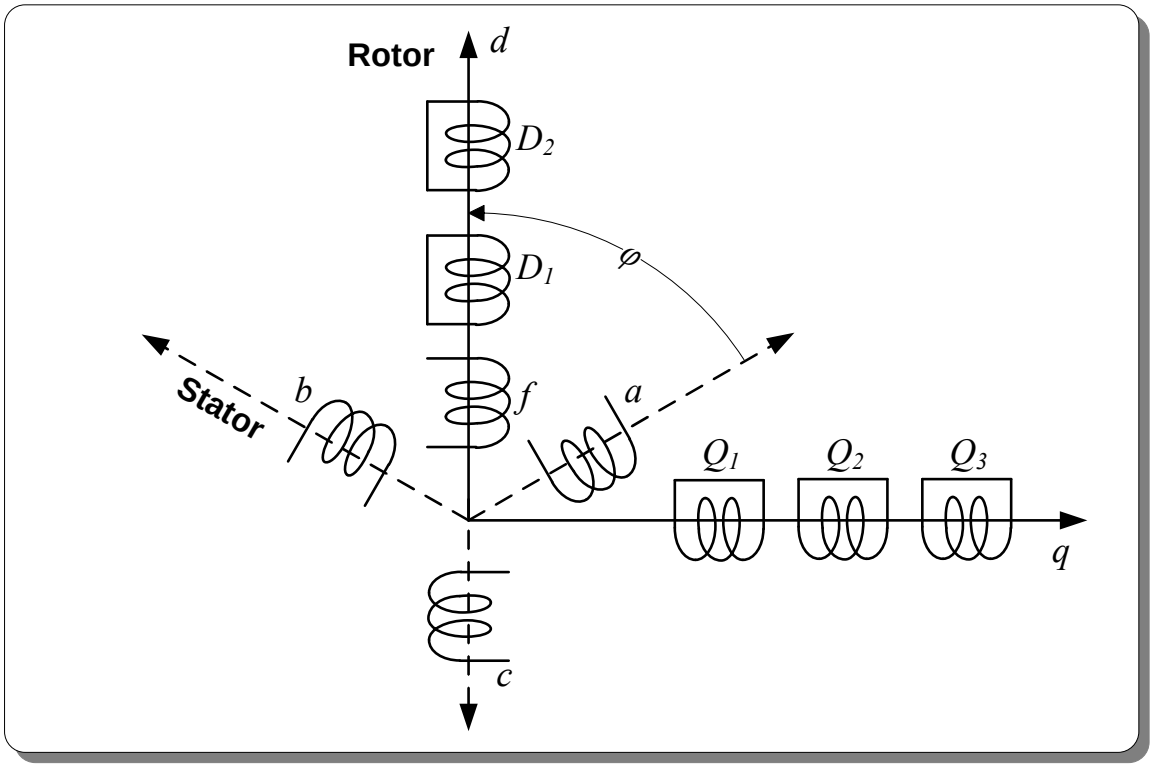

Figure 3.13: Windings in UMM.

where $\boldsymbol{L}$ is the winding leakage inductance matrix given as

$$\boldsymbol{L} = \begin{bmatrix} L_d & 0 & 0 & M_{df} & M_{dD_1} & M_{dD_2} & 0 & 0 & 0 \\ 0 & L_q & 0 & 0 & 0 & 0 & M_{qQ_1} & M_{qQ_2} & M_{qQ_3} \\ 0 & 0 & L_0 & 0 & 0 & 0 & 0 & 0 & 0 \\ M_{df} & 0 & 0 & L_f & M_{fD_1} & M_{fD_2} & 0 & 0 & 0 \\ M_{dD_1} & 0 & 0 & M_{fD_1} & L_{D_1} & M_{D_1D_2} & 0 & 0 & 0 \\ M_{dD_2} & 0 & 0 & M_{fD_2} & M_{D_1D_2} & L_{D_2} & 0 & 0 & 0 \\ 0 & M_{qQ_1} & 0 & 0 & 0 & 0 & L_{Q_1} & M_{Q_1Q_2} & M_{Q_1Q_3} \\ 0 & M_{qQ_2} & 0 & 0 & 0 & 0 & M_{Q_1Q_2} & L_{Q_2} & M_{Q_2Q_3} \\ 0 & M_{qQ_3} & 0 & 0 & 0 & 0 & M_{Q_1Q_3} & M_{Q_2Q_3} & L_{Q_3} \end{bmatrix} \tag{3.46}$$

with L and M standing for the self and mutual inductances respectively. In (3.44), the vectors of voltages $\boldsymbol{v}_{dq0}$, currents $\boldsymbol{i}_{dq0}$ and speed voltages $\boldsymbol{u}$ of the windings are expressed as

$$\begin{aligned} \boldsymbol{v}_{dq0} &= [\ v_d\ ,\ v_q\ ,\ v_0,\ v_f,\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ ], \\ \boldsymbol{i}_{dq0} &= [\ i_d\ ,\ i_q\ ,\ i_0,\ i_f,\ i_{D_1},\ i_{D_2},\ i_{Q_1},\ i_{Q_2},\ i_{Q_3}], \\ \boldsymbol{u} &= [\ -\omega\lambda_q,\ \omega\lambda_d,\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ ,\ 0\ ]; \end{aligned}$$

the winding resistance matrix $\boldsymbol{R}$ is a diagonal matrix, given as

$$\boldsymbol{R} = diag[R_d, R_q, R_0, R_f, R_{D_1}, R_{D_2}, R_{Q_1}, R_{Q_2}, R_{Q_3}];$$

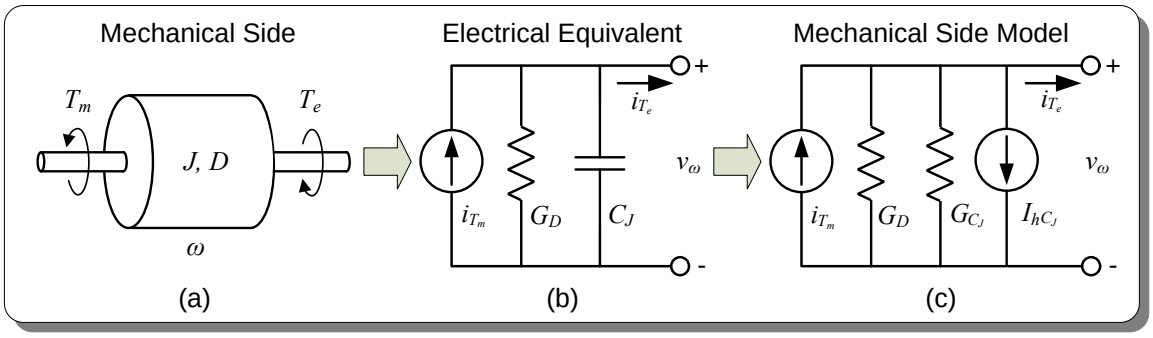

Figure 3.14: Electrical model of the mechanical part of UMM.

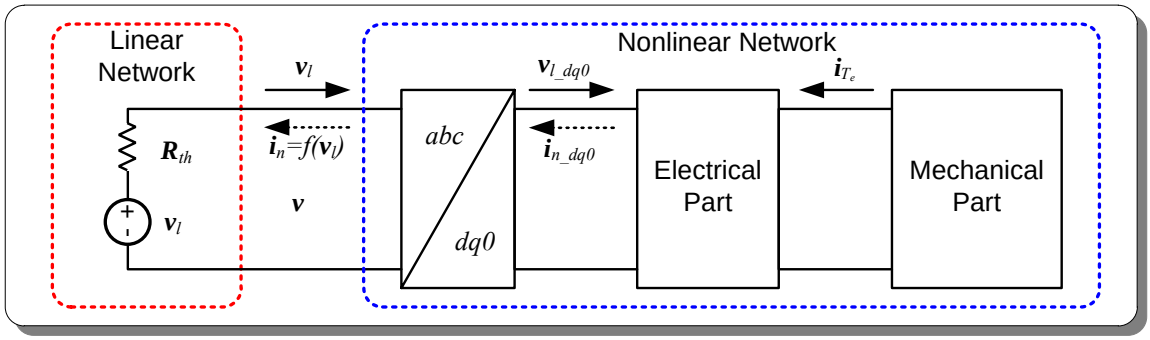

Figure 3.15: Interfacing UMM with the linear network via CMI.

and the history term $\boldsymbol{V}_h$ using Trapezoidal discretization can be expressed as

$$\boldsymbol{V}_h(t-\Delta t) = -\boldsymbol{v}_{dq0}(t-\Delta t) - \boldsymbol{R}\boldsymbol{i}_{dq0}(t-\Delta t) + \frac{2}{\Delta t}\boldsymbol{\lambda}_{dq0}(t-\Delta t) + \boldsymbol{u}(t-\Delta t). \tag{3.47}$$

The Park's transformation links the *abc* phase domain with the *dq0* rotating reference domain for any vector given as

$$\boldsymbol{x}_{dq0} = \boldsymbol{P}\boldsymbol{x}_{abc}, \tag{3.48}$$

where $\boldsymbol{P}$ is an orthogonal matrix defined as

$$\boldsymbol{P} = \sqrt{\frac{2}{3}} \begin{bmatrix} cos(\varphi) & cos(\varphi - \frac{2\pi}{3}) & cos(\varphi + \frac{2\pi}{3}) \\ sin(\varphi) & sin(\varphi - \frac{2\pi}{3}) & sin(\varphi + \frac{2\pi}{3}) \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}, \tag{3.49}$$

where φ denotes the rotor angle.

3.3.3.1.2 Mechanical Part The dynamics of the rotor can be described with the differential equation

$$T_m = J\frac{d\omega}{dt} + D\omega + T_e, \tag{3.50}$$

where T_m, T_e, J, D and ω denote the mechanical torque, electromagnetic torque, inertia, damping and rotor speed respectively, as shown in Fig. 3.14(a). Using the machine current $\boldsymbol{i}_{dq0}$, the electromagnetic torque T_e is calculated as

$$T_e = \lambda_d i_q - \lambda_q i_d. \tag{3.51}$$

Instead of the mass-shaft system, the mechanical part described in (3.50) is represented as a linear electrical equivalent circuit in the UMM as shown in Fig. 3.14(b). The equivalent differential equation of (3.50) is replaced by

$$i_{T_m} = C_J \frac{dv_\omega}{dt} + G_D v_\omega + i_{T_e}. \tag{3.52}$$

Discretizing the lumped equivalent capacitance C_J, the mechanical side model is shown in Fig. 3.14(c), where the equivalent conductance G_{C_J} and history current I_{hC_J} are given by

$$G_{C_J} = 2C_J/\Delta t, \tag{3.53}$$

$$I_{hC_J}(t-\Delta t) = -I_{hC_J}(t-2\Delta t) - 2G_{C_J} v_\omega(t-\Delta t), \tag{3.54}$$

with the equivalent voltage v_ω is expressed as

$$v_\omega(t) = \frac{i_{T_m}(t) - i_{T_e}(t) - I_{hC_J}(t-\Delta t)}{G_D + G_{C_J}}. \tag{3.55}$$

Since the UMM is a nonlinear model which connects to the linear network, the compensation method [29] is used to circuit interface it with the EMT network solution. As shown in Fig. 3.15, the open-circuit node voltage of the nonlinear component $\boldsymbol{v}_l$, which is also the Thévenin equivalent voltage of the linear network, is first solved. Considering $\boldsymbol{v}_l$ as the input to the nonlinear component, the reaction current $\boldsymbol{i}_n$ from the nonlinear system can be calculated by the relational function f between $\boldsymbol{v}_l$ and $\boldsymbol{i}_n$. Injecting $\boldsymbol{i}_n$ into the linear network, the node voltage $\boldsymbol{v}$ of nonlinear component after compensation is given as

$$\boldsymbol{v} = \boldsymbol{v}_l + \boldsymbol{r}_{th}\boldsymbol{i}_n, \tag{3.56}$$

where $\boldsymbol{r}_{th}$ is the Thevenin equivalent resistance of the linear network looking into the open port from the nonlinear side. The mathematical nonlinearity involving the product of fluxes and currents in (3.51) is handled by an iteration method as shown in Fig. 3.16. Once the speed ω has converged, the currents $\boldsymbol{i}_{n\_dq0}$ are transferred back to the phase-domain as the incident currents $\boldsymbol{i}_n$ from the nonlinear network to the linear network.

3.3.3.2 Massive-thread parallel implementation

As shown in Fig. 3.17, the designed parallel module for the UMM includes 3 kernels within 3 stages. Stage 1 predicts the rotor speed ω_p and transfers the phase-domain inputs into

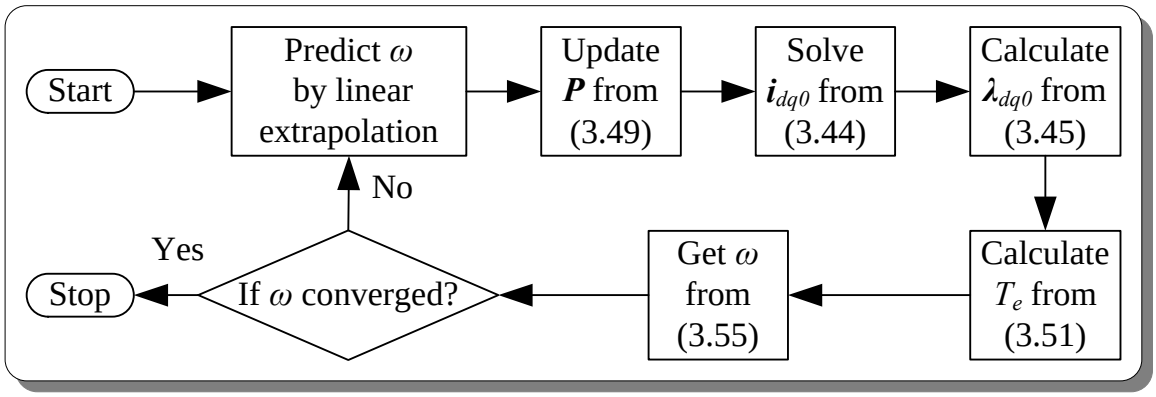

Figure 3.16: Iteration flow for UMM.

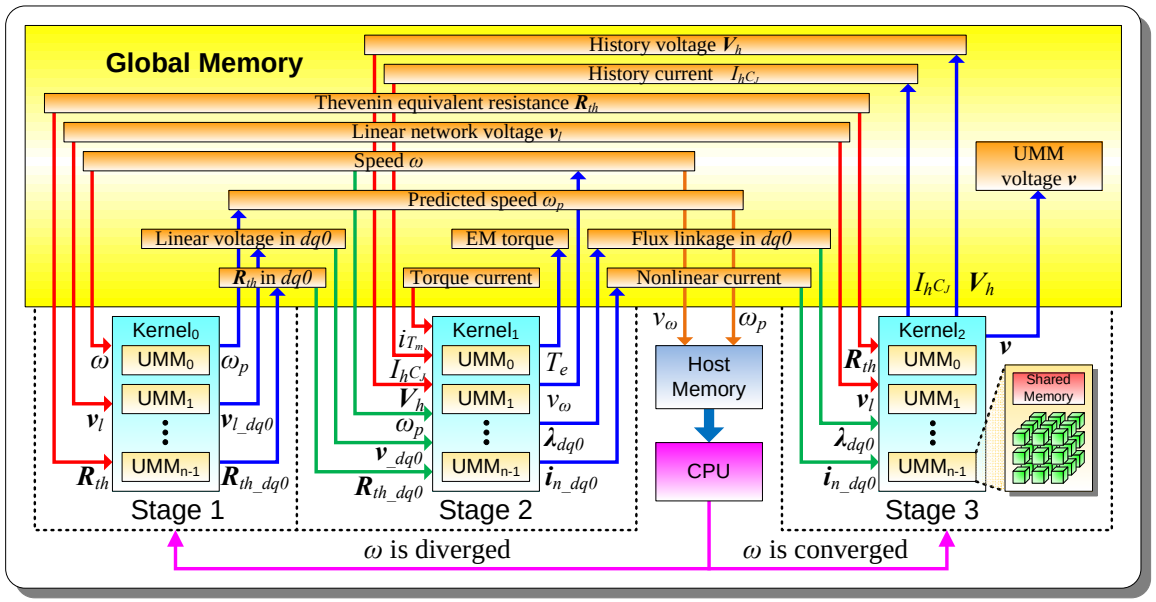

Figure 3.17: Massive-threading parallel UMM.

dq0 reference domain. Stage 2 is responsible for the computations of electrical part and mechanical part, and gets the electromagnetic torque T_e, the nonlinear current $\boldsymbol{i}_{n\_dq0}$ in *dq0* and the equivalent speed voltage v_ω. Before proceeding to Stage 3, the convergence of the rotor speed ω for all UM units is determined by the CPU to avoid the synchronous, efficient and random memory access issues arising from the parallel determination, and Stage 1 to 2 are repeated until all UMM units are converged or the maximum number of iterations are reached. Finally, Stage 3 updates the history variables and completes the calculation of the integrated UMM voltage $\boldsymbol{v}$. Similar to the ULM module, each UMM unit occupies a CUDA block running in SIMT, in which multiple threads are assigned to handle the vector and matrix operations based on SIMD, according to their dimensions. Shared memory inside the CUDA block is used for critical memory access during the vector and matrix operations.

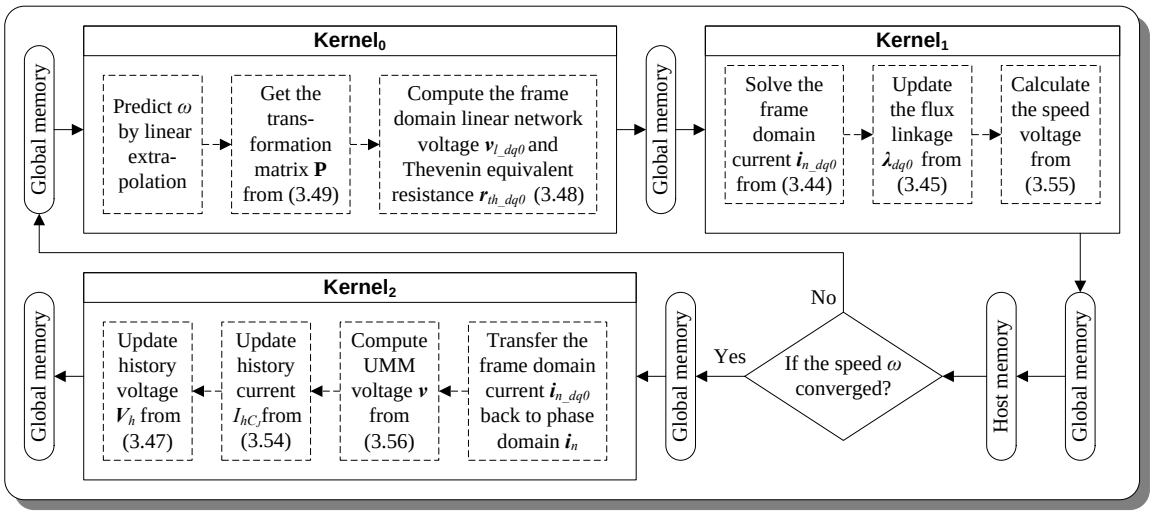

Figure 3.18: Kernel operation flow in the UMM.

Fig. 3.18 shows the operation flow in the kernels of the UMM module. In order to reduce the extra cost for kernels switch of CUDA program, as many as possible tasks are contained in a kernel unless the configuration (threads and memory) of the kernel has to be changed. Inside the $\textbf{Kernel}_0$, the rotor speed ω_p is predicted by extrapolation first, then the Park's transformation matrix $\boldsymbol{P}$ is updated from (3.49) to transfer the linear network voltages $\boldsymbol{v}_l$ and Thévenin equivalent resistance $\boldsymbol{R}_{th}$ into the variables $\boldsymbol{v}_{l\_dq0}$ and $\boldsymbol{R}_{th\_dq0}$ using (3.48). The $\textbf{Kernel}_1$ first solves the linear system using LU decomposition and forward-backward substitution from (3.44) to get the frame domain currents $\boldsymbol{i}_{n\_dq0}$. Then the flux linkages $\boldsymbol{\lambda}_{dq0}$ from (3.45) are updated, and finally the equivalent speed voltage v_ω is calculated using (3.55). In $\textbf{Kernel}_2$, the reference domain currents $\boldsymbol{i}_{n\_dq0}$ are transferred back to the phase-domain current $\boldsymbol{i}_n$ with the Park's transformation matrix $\boldsymbol{P}^{-1}$ based on the converged rotor speed ω; then the UMM voltages $\boldsymbol{v}$ is computed from (3.56) with the linear network voltages $\boldsymbol{v}_l$, Thévenin equivalent resistance $\boldsymbol{R}_{th}$ and the incident currents $\boldsymbol{i}_n$; finally, the history current I_{hC_J} and the history voltages $\boldsymbol{V}_h$ are updated from (3.54) and (3.47) respectively for the next time-step.

3.3.4 Newton-Raphson Iteration

3.3.4.1 Method formulation

As a numerical algorithm with quadratic convergence, the Newton-Raphson iteration method is used pervasively to solve the nonlinear problems in many areas as well as in EMTP. For the nonlinear equation

$$f(x) = 0, \tag{3.57}$$

the derivative of $f(x)$ at x_n can be approximated by the difference given as

$$f'(x_n) = \frac{f(x_n) - f(x_{n+1})}{x_n - x_{n+1}}. \tag{3.58}$$

When the iteration is converged, the approximate value is

$$f(x_{n+1}) \approx f(x) = 0. \tag{3.59}$$

Substituting (3.59) into (3.58), the recurrence formula is gotten as

$$f'(x_n)(x_n - x_{n+1}) = f(x_n). \tag{3.60}$$

For the nonlinear system functions $\boldsymbol{F}(\boldsymbol{x})$, the solution can be found by solving following linear system iteratively as

$$\boldsymbol{J}_{\boldsymbol{x}}^{\boldsymbol{F}}(\boldsymbol{x}_n - \boldsymbol{x}_{n+1}) = \boldsymbol{F}(\boldsymbol{x}_n), \tag{3.61}$$

where the Jacobian matrix $\boldsymbol{J}_{\boldsymbol{F}}$ are the partial derivatives of $\boldsymbol{F}(\boldsymbol{x})$ defined as

$$\boldsymbol{J}_{\boldsymbol{x}}^{\boldsymbol{F}} = \begin{bmatrix} \dfrac{\partial F_1}{\partial x_1} & \cdots & \dfrac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial F_n}{\partial x_1} & \cdots & \dfrac{\partial F_n}{\partial x_n} \end{bmatrix}. \tag{3.62}$$

Since the CMI is used to link the linear network and nonlinear components, the compensation equations are expressed in (3.2). Assuming the current and voltage relations of nonlinear components is

$$\boldsymbol{v}_{km} = \boldsymbol{f}(\boldsymbol{i}_{km}), \tag{3.63}$$

the nonlinear system functions are gotten as

$$\boldsymbol{F}(\boldsymbol{i}_{km}) = \boldsymbol{f}(\boldsymbol{i}_{km}) - \boldsymbol{v}_{kmo} + \boldsymbol{R}_{eq}\boldsymbol{i}_{km}. \tag{3.64}$$

Therefore, the Jacobian matrix is

$$\boldsymbol{J}_{\boldsymbol{i}_{km}}^{\boldsymbol{F}} = \boldsymbol{J}_{\boldsymbol{i}_{km}}^{\boldsymbol{f}} + \boldsymbol{R}_{eq}, \tag{3.65}$$

where $\boldsymbol{J}_{\boldsymbol{i}_{km}}^{\boldsymbol{f}}$, denoting the Jacobian matrix of $\boldsymbol{f}(\boldsymbol{i}_{km})$, is updated in every iteration using (3.62). The norms of $\Delta\boldsymbol{i}_{km}$ and $\boldsymbol{F}(\boldsymbol{i}_{km})$ are used as the convergence criteria given as

$$\|\boldsymbol{i}_{km}^{n+1} - \boldsymbol{i}_{km}^{n}\| < \epsilon_1 \tag{3.66a}$$

$$\|\boldsymbol{F}(\boldsymbol{i}_{km}^{n+1})\| < \epsilon_2 \tag{3.66b}$$

where ϵ_1 and ϵ_2 are sufficiently small values.

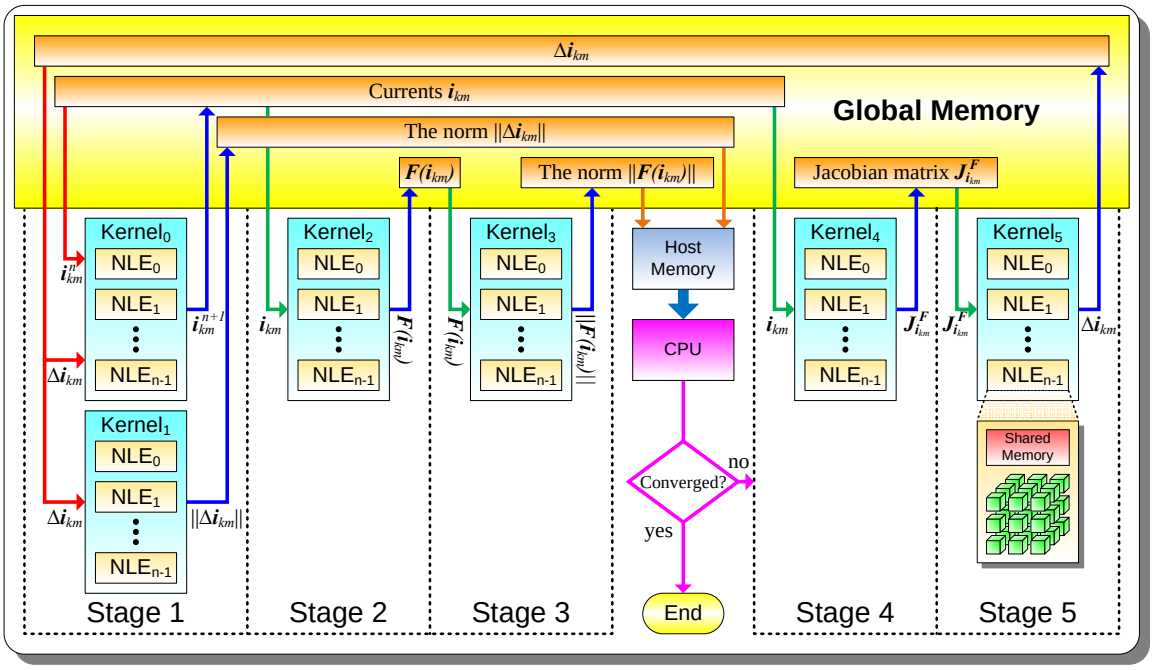

Figure 3.19: Massive-threading parallel NRIM.

3.3.4.2 Massive-thread parallel implementation

As shown in Fig. 3.19, the designed parallel module for NRI has 6 kernels divided into 5 stages. Stage 1 includes 2 kernels which update $\boldsymbol{i}_{km}$ and calculate the norm of $\boldsymbol{\Delta i}_{km}$ respectively. Stage 2 calculates the right hand side of (3.61). Stage 3 calculates the norm of $\boldsymbol{F}(\boldsymbol{i}_{km})$. With the norms calculated in Stage 1 and Stage 3, the convergence is tested by the criteria given in (3.66). If it is *True*, the iteration is terminated with the current results of $\boldsymbol{i}_{km}$; if it is *False*, Stage 4 will be processed. Stage 4 updates the Jacobian matrix of $\boldsymbol{F}(\boldsymbol{i}_{km})$. Stage 5 gets the new $\boldsymbol{\Delta i}_{km}$ by solving the linear system. Then the whole procedure will be repeated until the results are converged or the maximum loop limitation is reached. Each nonlinear element (NLE) is assigned to a CUDA block running in SIMT , in which multiple threads are assigned to handle the vector and matrix operations based on SIMD, according to their dimensions. In 3-phase circuit, the dimension of nonlinear equations is 3 typically, so that the number of CUDA threads per block will be 9, which is far less than the limitation of the threads per block. Shared memory is also involved in most vector and matrix operations to reduce the access delay.

The operation flow of NRIM is shown in Fig. 3.20. With the $\boldsymbol{\Delta i}_{km}$, the next step $\boldsymbol{i}_{km}$ is updated in **Kernel**$_0$ and the its norm are calculated in **Kernel**$_1$ from (3.66a). The **Kernel**$_2$ updates the voltages between k-end and m-end from (3.63), and then updates the right hand side vector $\boldsymbol{F}(\boldsymbol{i}_{km})$ of the linear system in (3.61). The **Kernel**$_3$ calculates the norm of $\boldsymbol{F}(\boldsymbol{i}_{km})$. When the two norms of the convergence criteria are obtained, the process direction is determined by them. If the results are not converged, the Jacobian matrix of $\boldsymbol{f}(\boldsymbol{i}_{km})$ is

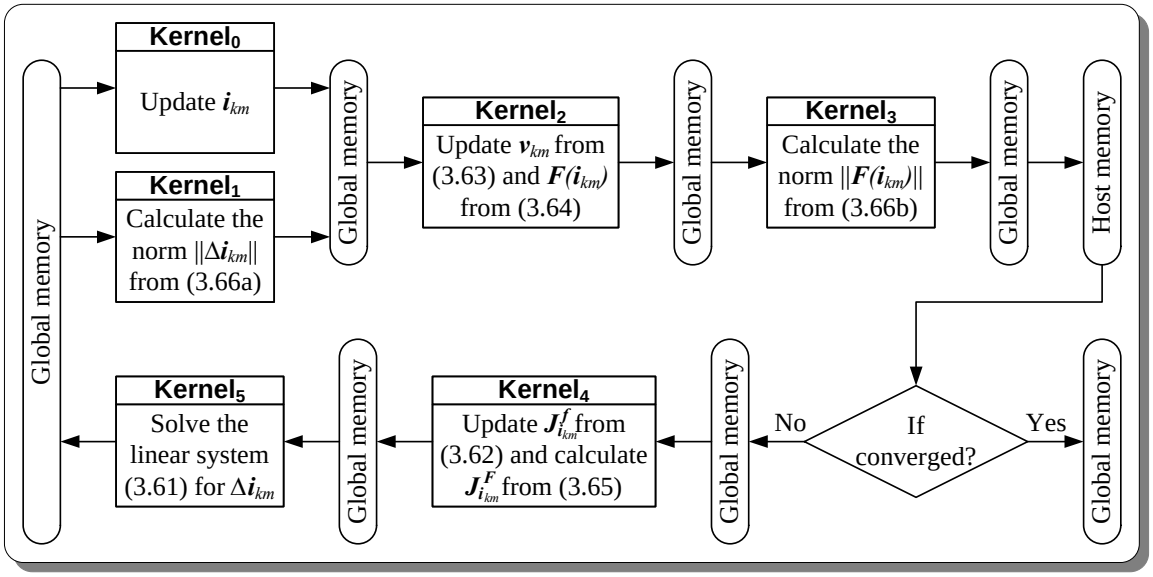

Figure 3.20: Kernel operation flow in the NRIM.

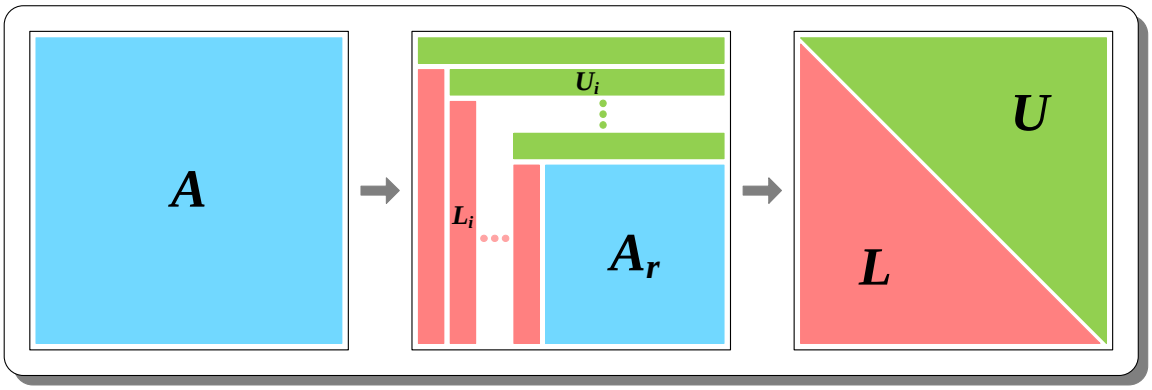

Figure 3.21: LU decomposition.

updated from (3.62), and then the Jacobian matrix of $\boldsymbol{F}(\boldsymbol{i}_{km})$ is updated from (3.65) in the **Kernel**$_4$. The **Kernel**$_5$ solves the linear system (3.61) to obtain the new $\boldsymbol{\Delta i}_{km}$, which are the new input to **Kernel**$_0$ and **Kernel**$_1$. The iteration keeps going until the convergence criteria is satisfied or the maximum loop limitation is reached.

3.3.5 Forward-Backward Substitution with LU Factorization

3.3.5.1 Method formulation

In order to solve the n-order system of linear equations

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{3.67}$$

in the EMT simulation, a massive-threading parallel direct, non-iterative, linear solver was developed. Although iterative methods such as the conjugate gradient algorithm [36] have

been implemented on the GPU, the parallelism is only available within a single iteration. Since the next iteration depends on the previous one, iterations cannot be processed in parallel although the same algorithm is used within the iterations. Moreover, since the number of iterations determined by the convergence of each matrix is dissimilar even for matrices of the same dimension, extra considerations are necessary to synchronize with other parallel tasks. Compared with iterative methods, direct methods can give conformable solutions with multielement parallelism and stable solution time regardless of the elements in the matrix, although at the cost of higher algorithmic complexity. In the literature GPU accelerated direct solvers have been proposed for symmetric sparse matrices using Cholesky decomposition [37], and multifrontal computations [38], showing a significant speedup. However, since the matrices in EMT simulation are not all symmetrical, a parallel direct method is applied for an efficient solution.

As shown in Fig. 3.21, partial pivoting LU factorization decomposes the matrix A into the product of lower and upper triangular matrices L and U given as

$$\boldsymbol{PA} = \boldsymbol{LU}, \tag{3.68}$$

where $\boldsymbol{P}$ is a permutation matrix for solution stability formed by exchanging the rows of $\boldsymbol{A}$. The element of $\boldsymbol{L}$ in (3.68) is calculated by

$$\boldsymbol{L}[i][k] = \boldsymbol{A}[i][k]/\boldsymbol{A}[i][i] \qquad (0 < i < n, i < k < n). \tag{3.69}$$

The element of $\boldsymbol{U}$ in (3.68) can be calculated by updating the rest part of $\boldsymbol{A}$ expressed as

$$\boldsymbol{U}[j][k] = \boldsymbol{A}[j][k] - \boldsymbol{A}[j][i] * \boldsymbol{L}[i][k] \qquad (0 < i < n, i < j < n, i < k < n). \tag{3.70}$$

Substituting (3.68) to (3.67), the linear system become

$$\boldsymbol{LUx} = \boldsymbol{Pb}. \tag{3.71}$$

Defining

$$\boldsymbol{Ux} = \boldsymbol{y}, \tag{3.72}$$

we get

$$\boldsymbol{Ly} = \boldsymbol{Pb}. \tag{3.73}$$

Denoting $\boldsymbol{b_P} = \boldsymbol{Pb}$, the interim vector $\boldsymbol{y}$ can be solved from (3.73) firstly; then the final solution $\boldsymbol{x}$ can be solved from (3.72). Since $\boldsymbol{L}$ is lower triangular, forward substitution is used to solve $\boldsymbol{y}$ easily, given as

$$\begin{cases} \boldsymbol{y}[i] = \boldsymbol{b_P}[i]/\boldsymbol{L}[i][i] \\ \boldsymbol{y}[j] = \boldsymbol{b_P}[j] - \boldsymbol{L}[j][i] * \boldsymbol{y}[i] \end{cases} \qquad (0 \leqslant i < n, i < j < n); \tag{3.74}$$

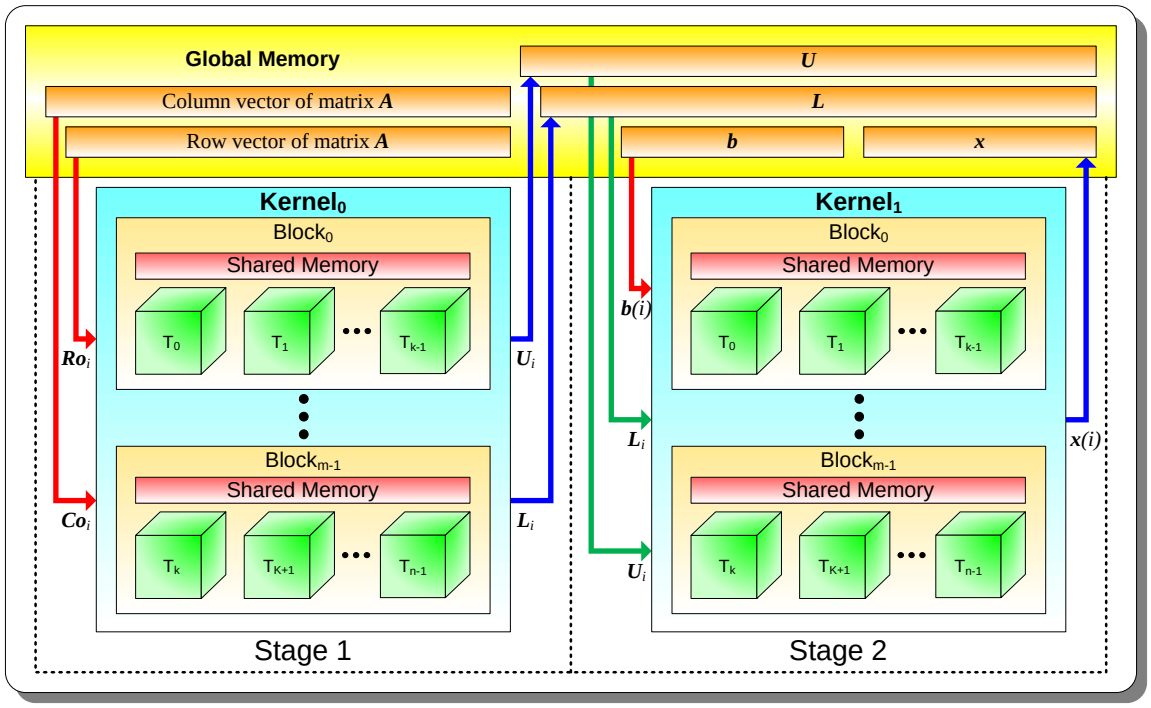

Figure 3.22: Massive-threading parallel LU-FBSM.

similarly, the solution $\boldsymbol{x}$ are solved by backward substitution directly, given as

$$\begin{cases} \boldsymbol{x}[i] = \boldsymbol{y}[i]/\boldsymbol{U}[i][i] \\ \boldsymbol{x}[j] = \boldsymbol{y}[j] - \boldsymbol{U}[j][i] * \boldsymbol{x}[i] \end{cases} \qquad (n > i \geqslant 0, n > j > i). \tag{3.75}$$

Different from Gaussian elimination, $\boldsymbol{L}$ and $\boldsymbol{U}$ do not need to be recomputed unless the system $\boldsymbol{A}$ are changed.

3.3.5.2 Massive-thread parallel implementation

The massive-threading parallel LU & forward-backward substitution module (LU-FBSM), shown in Fig. 3.22, consists of 2 CUDA kernels within 2 stages. Stage 1 decomposes the matrix A into lower and upper triangular matrices, L and U, by LU factorization. And then Stage 2 solves the linear system by forward and backward substitutions with L and U. Both stages include internal iterations to traverse all columns and rows of the matrices. If the dimension of the matrix n exceeds the number of threads per block, the elements per column or row will be grouped into multiple CUDA blocks to be computed in parallel. Due to the internal loop, the number of which depends on the dimension of the matrix, shared memory can significantly improve the performance of LU decomposition and substitution by avoiding the data exchange between CUDA cores and global memory in every iteration.

The operation flow of LU-FBSM is shown in Fig. 3.23. In **Kernel**$_0$, the column of $\boldsymbol{L}$ is updated first from (3.69), then the elements of row of $\boldsymbol{U}$ and all remainder element of the

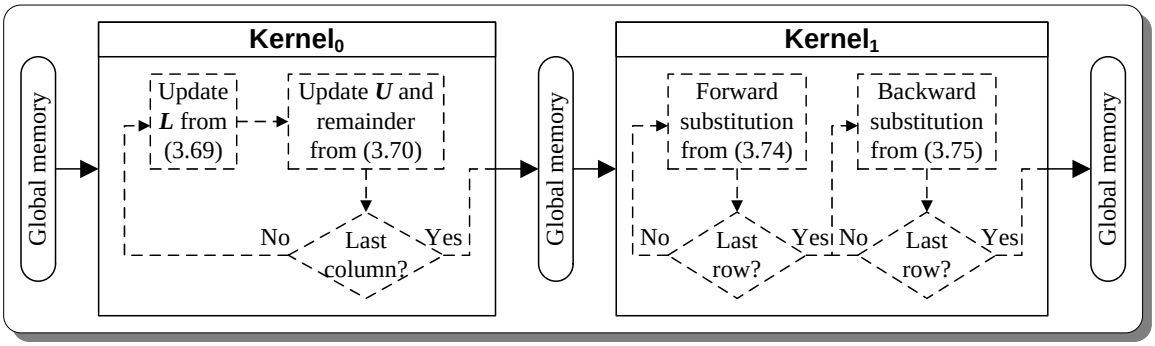

Figure 3.23: Kernel operation flow in LU-FBSM.

```
for all block nodes of size n{
    for i = 0 to n-1{
        find the max of column i;
        get the row id r_max;
        if A[i][r_max] ≠ A[i][i]
            swap row i and row r_max;
        update L;
        update U and remainder;
    }
}
```

CUDA Mapping

```
b = blockIdx.x;  n = blockDim.x;
r = threadIdx.x; c = threadIdx.y;
for(i=0;i<n-1;i++){
    if(A[i][r] == max(A[i]))
        r_max = r;
    if(A[i][r_max] != A[i][i])
        swap(A[c][i],A[c][r_max]);
    if(r>i)
        A[i][r] = A[i][r]/A[i][i];
    if(r>i&&c>i)
        A[c][r] -= A[c][i]*A[i][r];
}
```

Figure 3.24: LU decomposition CUDA mapping.

rest part of $\boldsymbol{A}$ are updated. The loop will not stop until all columns are updated to obtain $\boldsymbol{L}$ and $\boldsymbol{U}$, which are stored in an combined $n \times n$ matrix since there is no overlap between them, given as

$$\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{U} + \boldsymbol{I}, \tag{3.76}$$

where $\boldsymbol{I}$ is the identity matrix. In $\textbf{Kernel}_1$, the interim vector $\boldsymbol{y}$ is solved first by forward substitution from (3.74) using $\boldsymbol{L}$ and $\boldsymbol{b}$ with an n-step iteration; and then, backward substitution is applied to solve the final solution $\boldsymbol{x}$ from (3.75) using $\boldsymbol{U}$ and $\boldsymbol{y}$ with another n-step iteration scanning all rows of $\boldsymbol{U}$. All intermediate variables, vectors and matrices are stored in the shared memory during the iteration, and only the results of the kernels are transferred to global memory since the shared memory will be refreshed when the kernels are switches. The pseudo, shown in Fig. 3.24, gives the code mapping from normal serial C to parallel CUDA algorithm. Before partial pivoting, the maximum element of the column i of $\boldsymbol{A}$ is found as $\boldsymbol{A}[i][r_{max}]$, where r_{max} denotes the row index of the maximum element of the column i of $\boldsymbol{A}$. The binary scan operation [39] is applied to parallelize partial pivoting, which reduces the step complexity from $O(N)$ to $O(log_2 N)$. The LU factorization of a singular square matrix whose rank is less than its order can also be supported by this algorithm.

3.4 Summary

In this chapter, the parallel EMT simulator framework, including the hardware and software architectures; compensation method data interface; massive-threading parallel component modules, including ULEPM, ULM and UMM, and numerical solver modules, including NRIM and LU-FBSM, are elaborated from theoretical formulas to parallel implementation. In the design of the parallel EMT simulation system, the features and characteristics of GPU are sufficiently considered and applied to accelerate the computation of EMT simulation. The fully modularized design makes the whole EMT simulation system flexible and extensible, in which each module is developed, tested and updated independently.

4

Massive-thread Case Study and Data Analysis

4.1 Large-Scale EMT Simulation Case Study

Base on the parallel modules developed for various power system components, and numerical solvers in Chapter 3, the parallel massive-threading EMT program (MT-EMTP) is tested for simulating large-scale power systems in this chapter. Using the UMM, ULM and LPE to model a network, the nodal equation is given in (3.1). In general, $\boldsymbol{Y}$ is very sparse. For example, for the test system, the IEEE 39-bus power system, shown in Fig. 4.1, the original $\boldsymbol{Y}$ is shown in Fig. 4.2(a). It is 97.39% sparse with 357 nonzero elements. With increasing network size, the admittance matrix becomes even more sparse. It is considerably inefficient to handle a sparse matrix with traditional parallel dense algorithms, and the traditional parallel sparse algorithm is unsuitable for the GPU architecture; therefore, a specific sparse structure, the node mapping structure (NMS), which takes advantage of the component models, is proposed using a graph optimizing method, known as the block node adjustment (BNA), which does not affect the condition number of the matrix, to re-

Table 4.1: Hardware Specification

GPU		CPU	
Tesla^{TM} C2050 (Fermi)		AMD Phenom^{TM} II 955BE	
Cores	448	Cores	4
Frequency	1.15GHz	Frequency	3.2GHz
Global memory	3GB	System memory	16GB
CUDA Version	4.0	L2 Cache	2MB
CUDA Capability	2.0	L3 Cache	6MB

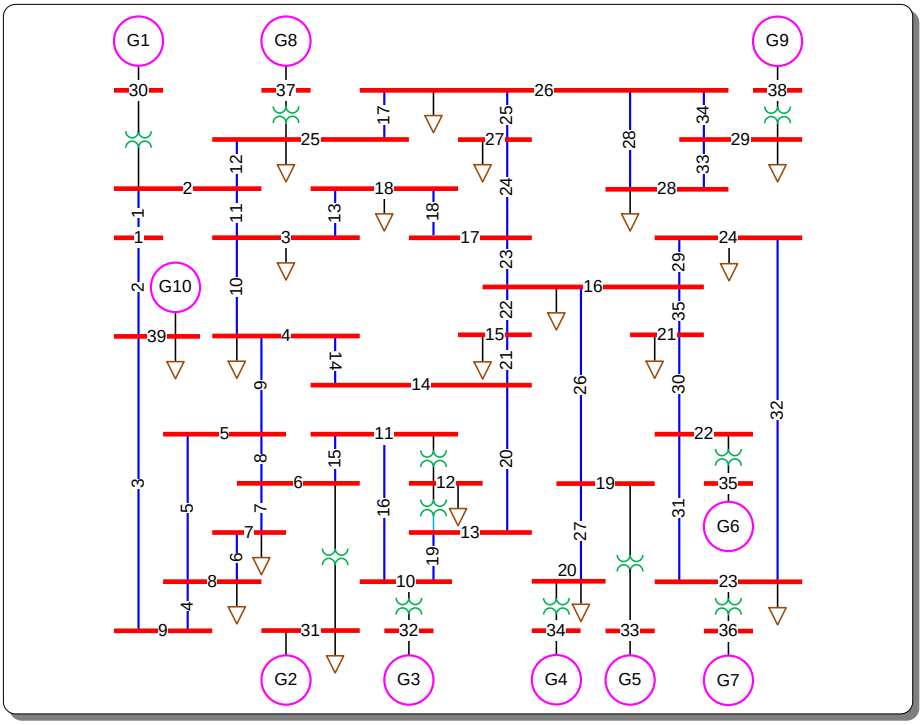

Figure 4.1: Single-line diagram of the IEEE 39-bus power system.

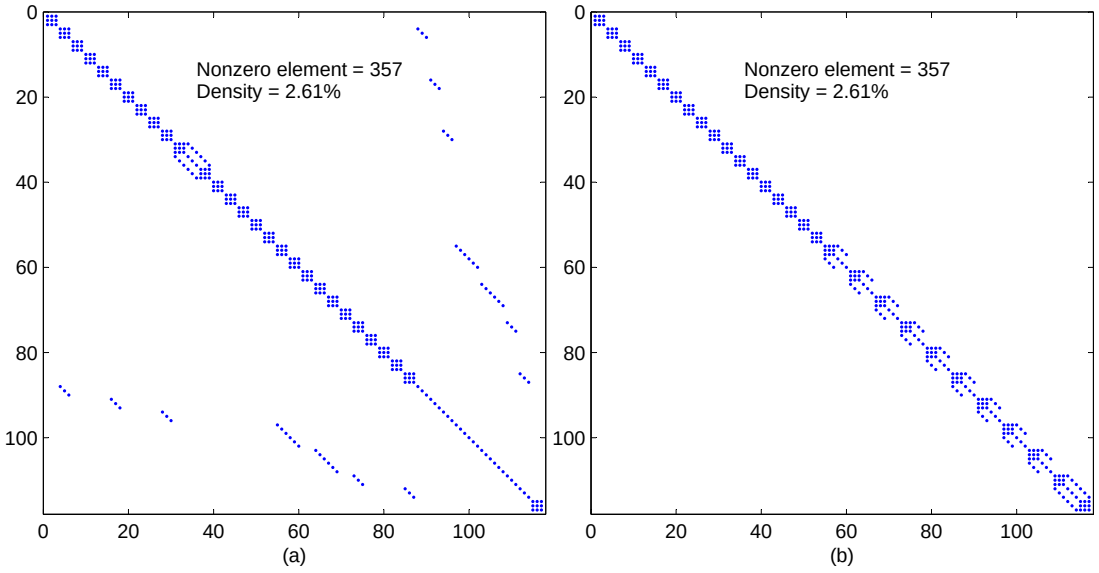

Figure 4.2: The pattern of Y matrix of IEEE 39-bus system. (a) Before block node adjustment (BNA); (b) After BNA.

shape the original $\boldsymbol{Y}$ matrix. In the BNA method, the minimal perfect hash [40] and integer sorting are applied to avoid string operations so that the complexity is reduced from $O(n^2)$ to $O(n)$.

The resulting matrix after BNA has a perfect block diagonal pattern as shown in Fig.

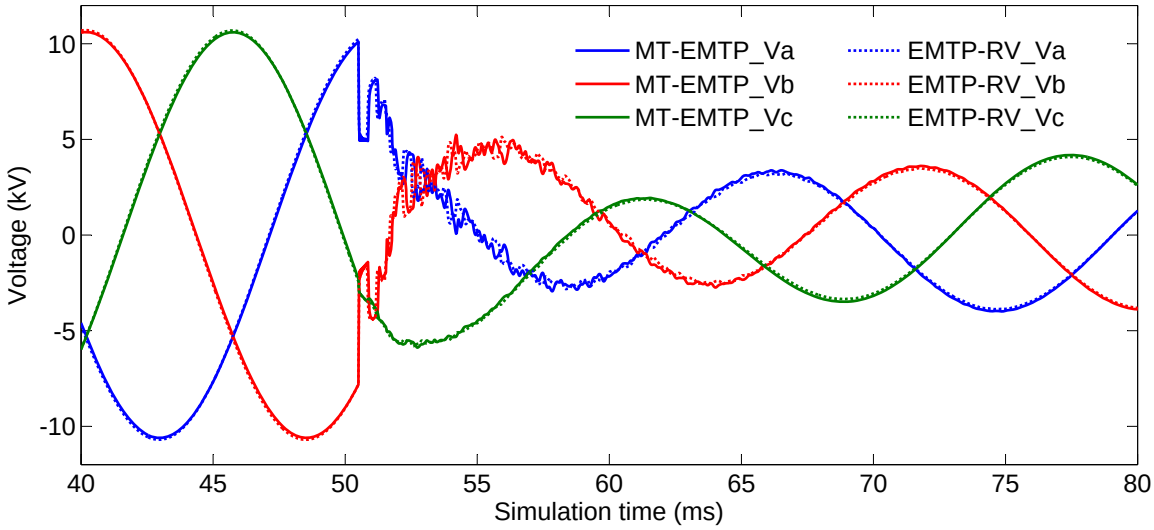

Figure 4.3: Comparison of simulation results (3-phase voltages) between MT-EMTP and EMTP-RV at Bus 5 during a 3-phase fault at Bus 4.

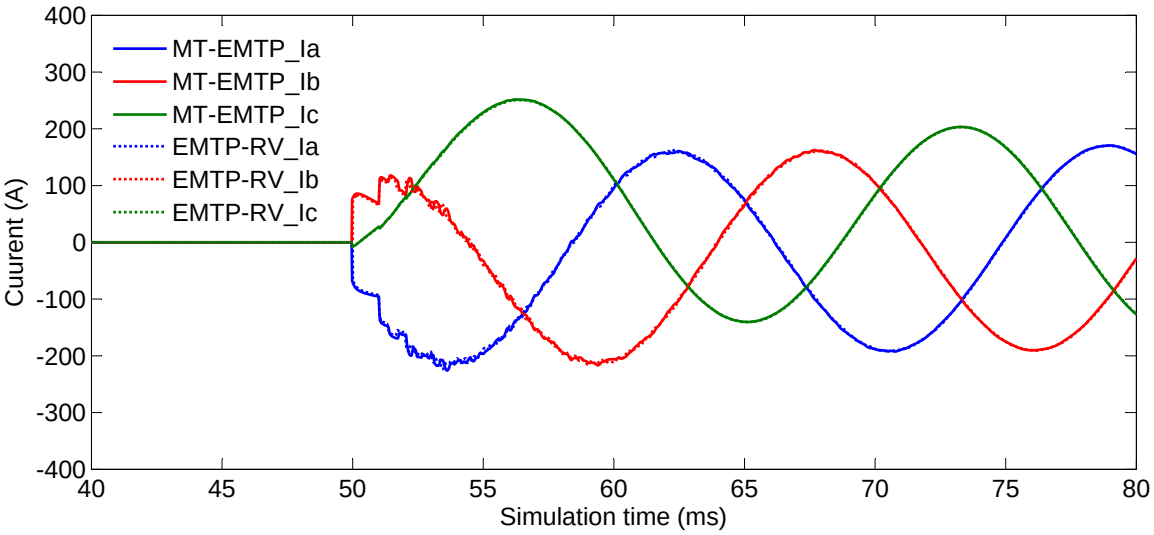

Figure 4.4: Comparison of simulation results (3-phase fault currents) between MT-EMTP and EMTP-RV at Bus 4.

4.2(b) where the number of blocks depends on the number of decoupled systems. Therefore, only the decoupled blocks in the admittance are stored in the host/device memory, which significantly reduces the pressure of data transfer for large-scale admittance matrices. Since all sub-systems can be computed independently, all blocks in the admittance matrix $\boldsymbol{Y}$ can be parallelized in the GPU. A decoupled sparse linear solver using LU-FBSM proposed in chapter 3 to compute the unknown node voltages.

The specifications of the hardware used are listed in the Table 4.1. The MT-EMTP (64-bit code) program was executed on the Fermi GPU, while EMTP-RV (32-bit code) was running on the AMD CPU, both using 64-bit double precision floating point data. The

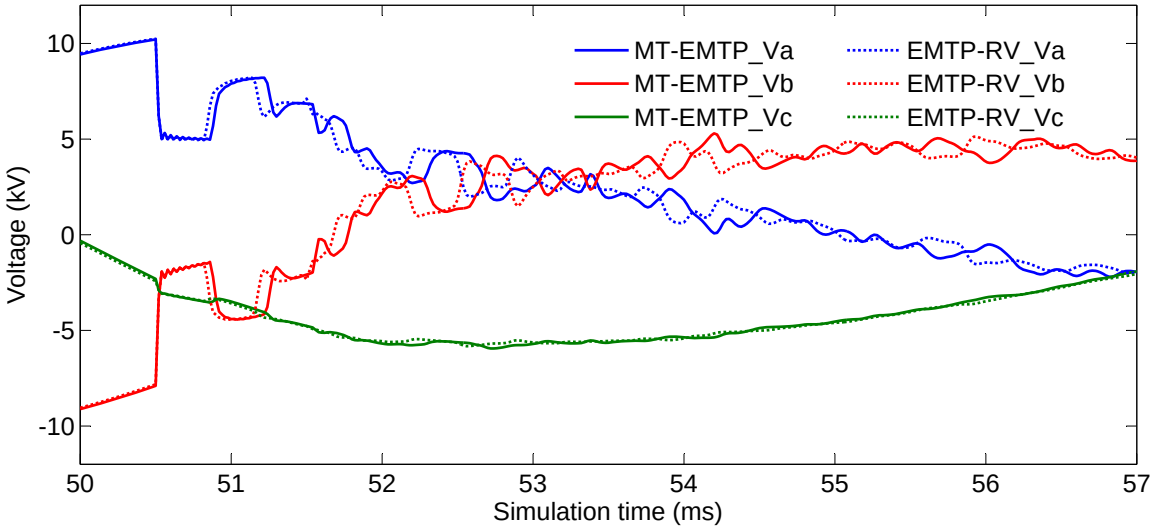

Figure 4.5: Zoomed-in view of Fig. 4.3 from $(t = 0.05s)$ to $(t = 0.057s)$.

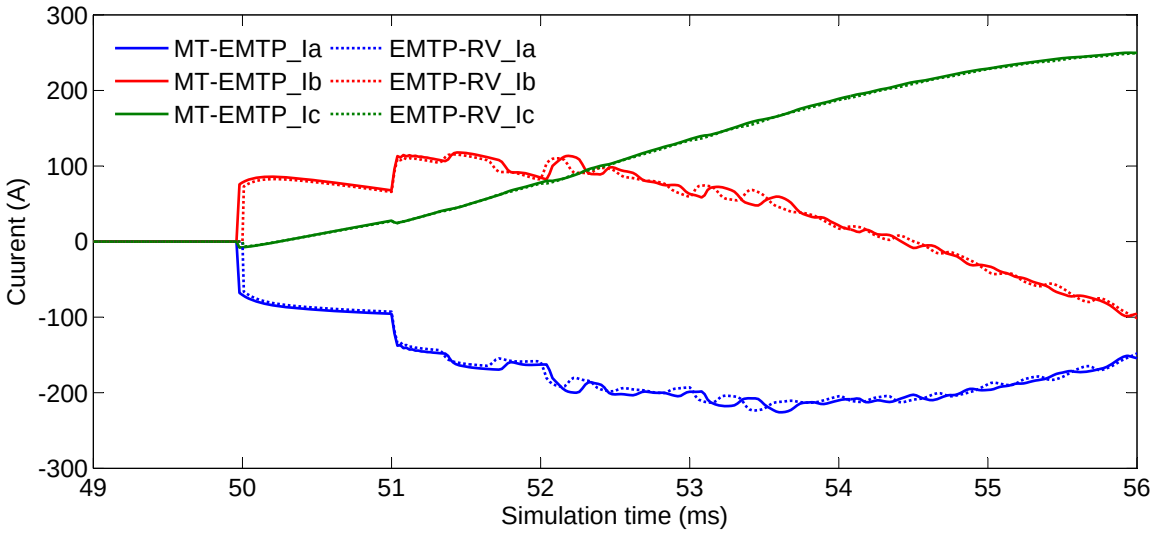

Figure 4.6: Zoomed-in view of Fig. 4.4 from $(t = 0.049s)$ to $(t = 0.056s)$.

simulation time is 100ms and the time-step is 20μs, and the total simulation steps are 5000. The CDA algorithm [28], in which the backward Euler rule replaces the Trapezoidal rule to discretize the differential and integral equations, are used to suppress the potential numerical oscillation.

4.1.1 Test case for a fault on the transmission line

As a regular fault in the test power system (Fig. 4.1), a 3-phase fault event occurs at Bus 4. The time-domain voltage waveforms at Bus 5 of the test power system are shown in Fig. 4.3. The fault currents at Bus 4 are shown in Fig. 4.4. The results from EMTP-RV and MT-EMTP are superimposed in figures 4.3 through 4.6, which show close agreement.

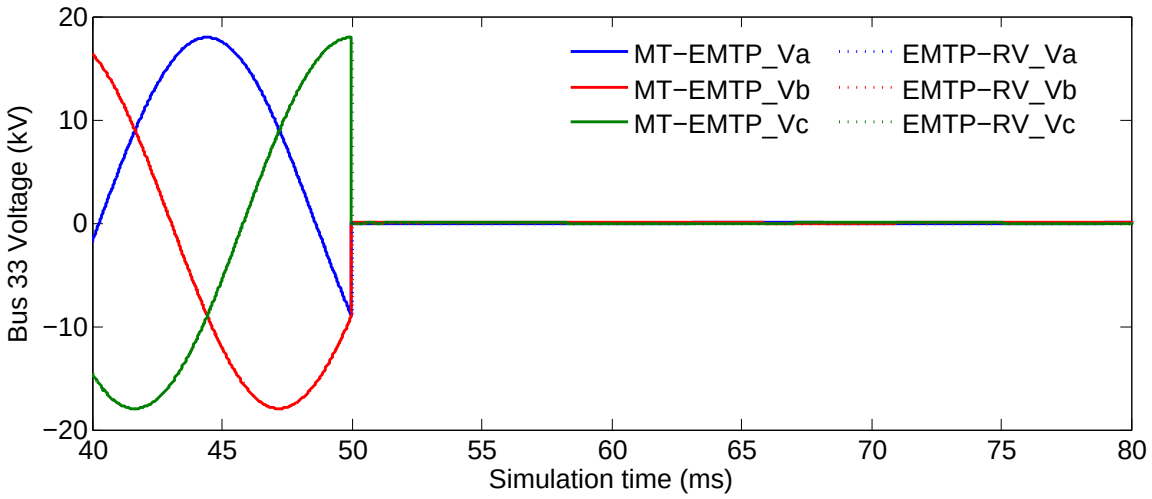

Figure 4.7: Comparison of simulation results (3-phase voltages) between MT-EMTP and EMTP-RV at Bus 33 during a 3-phase fault at Bus 33.

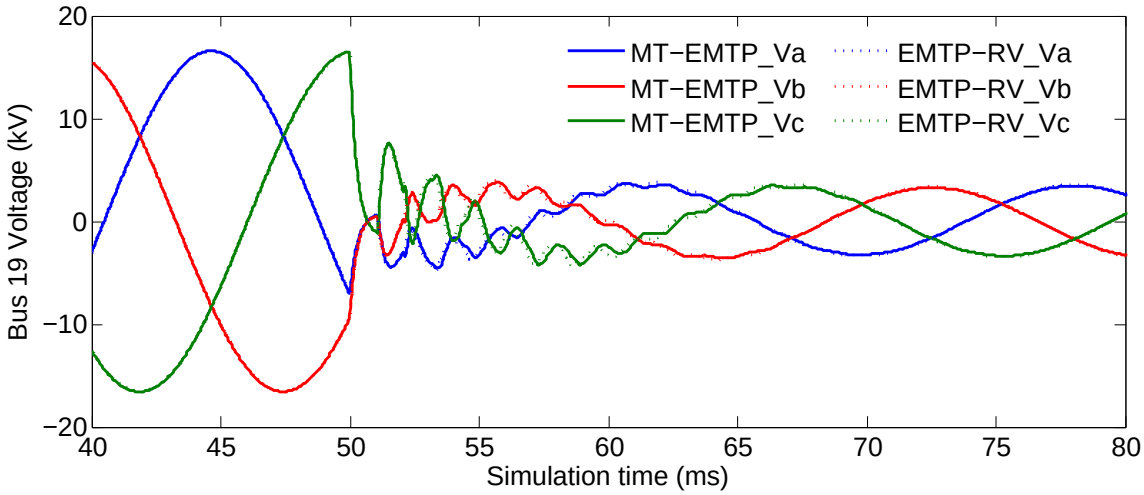

Figure 4.8: Comparison of simulation results (3-phase voltages) between MT-EMTP and EMTP-RV at Bus 19 during a 3-phase fault at Bus 33.

The zoomed-in figures, Fig. 4.5 and Fig. 4.6, show that there is a slight time-shift in the transients. The possible reasons for these differences could be different initial system parameters, modeling differences, and solution algorithm differences, for example, the use of compensation and CDA algorithms, in the MT-EMTP. It is known that even the mainstream commercial EMT simulation software will not give exactly identical results in many cases; and even for the same algorithm implemented by different computer languages, there may be some slight differences in the result. According to the zoomed in figures, the phase difference is about several time steps, which is $20\mu s$, the total difference is about $0.1ms$. It should be acceptable relating to the 60Hz frequency. The phase difference is not increasing over time because the two waveforms are synchronous in the next period in Fig.

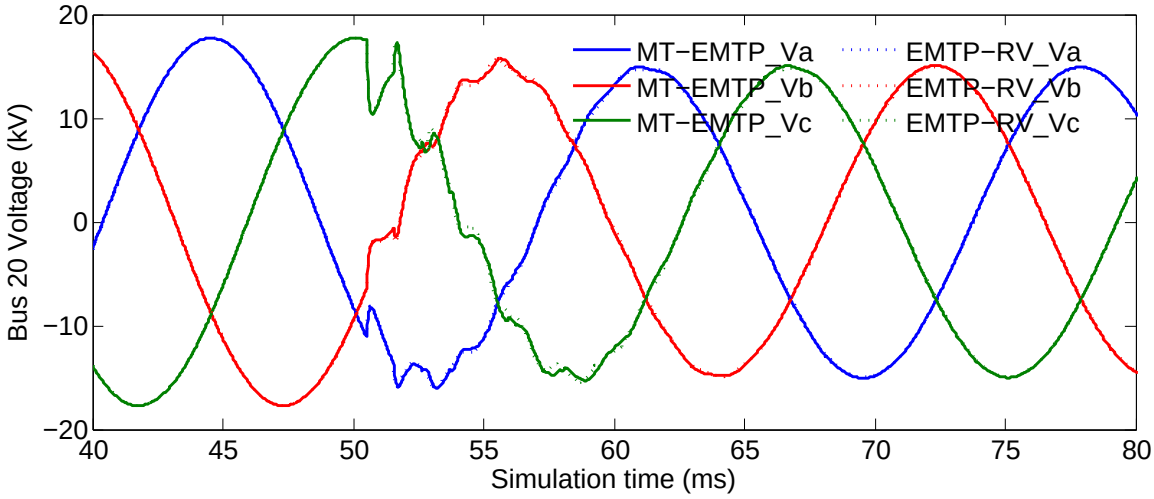

Figure 4.9: Comparison of simulation results (3-phase voltages) between MT-EMTP and EMTP-RV at Bus 20 during a 3-phase fault at Bus 33.

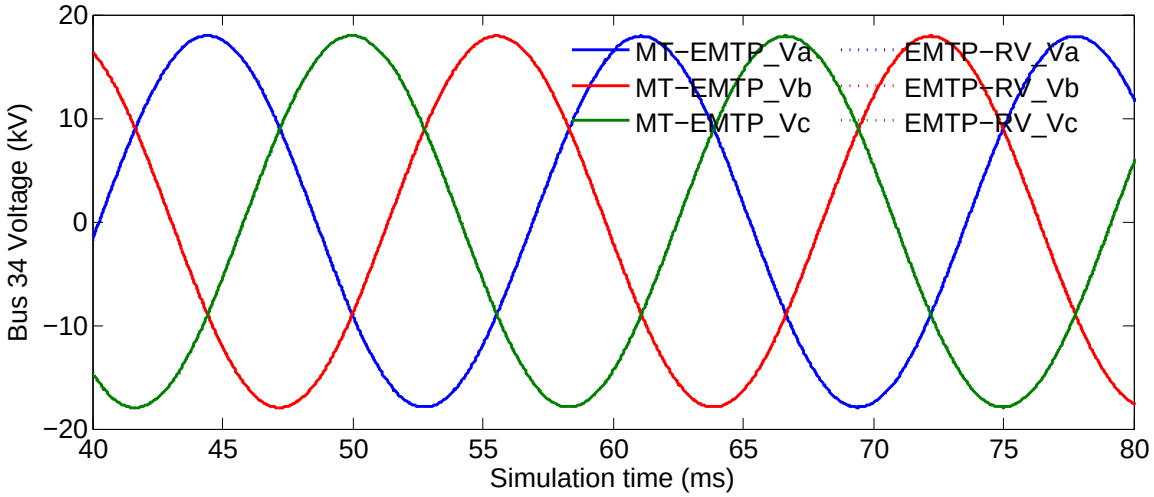

Figure 4.10: Comparison of simulation results (3-phase voltages) between MT-EMTP and EMTP-RV at Bus 34 during a 3-phase fault at Bus 33.

4.3 and Fig. 4.4.

4.1.2 Test case for a fault on the machine

In order to stress the source in the test power system, the 3-phase fault is set on Bus 33 which connects the synchronous machine G5 directly. The 3-phase voltages of Bus 33, Bus 19, Bus 20 and Bus 34 are compared with the outputs of the same test system in EMTP-RV, as shown in Fig. 4.7, Fig. 4.8, Fig. 4.9 and Fig. 4.10. From near to far, these four figures show the different degree of influence from the fault, and the comparisons give the closely matching results between two simulations. The difference only occurs in the transient

Table 4.2: Comparison of execution time for various system sizes between EMTP-RV® and GPU-based MT-EMTP for simulation duration 100ms with time-step 20μs

System structure (3-phase)					Execution time (s)		Speedup
Scale	Buses	Devices			EMTP-RV	MT-EMTP	
		LPEs	ULMs	UMMs			
1	40	31	35	10	0.967	0.875	1.11
2	79	61	72	20	2.012	1.542	1.30
4	157	121	148	40	4.118	2.529	1.63
8	313	241	300	80	9.625	4.583	2.10
16	625	481	608	160	25.988	8.031	3.24
32	1249	961	1224	320	65.567	14.900	4.40
48	1873	1441	1840	480	115.724	22.204	5.21
63	2458	1891	2425	630	168.502	29.056	5.75

duration of about $0.1ms$. As to the 60Hz signal, the differences are about 0.6% (less than 1%), which means the results coming from MT-EMTP are reasonable and acceptable and can reproduce the transients of the test system correctly.

4.1.3 Test case for the execution time on various scales of power systems

To evaluate computational efficiency, execution times of test systems of increasing size were recorded. Eight large-scale test cases were created by expanding the original IEEE 39-bus system with detailed modeling of all components. Sub-systems (39-bus) were interconnected with the systems around them by 2 additional transmission lines. The execution times are shown in the Table 4.2, which also includes the number of buses and devices in the systems. All the lines in these test cases were modeled using ULM and the machines using UMM. As can been seen, when the system size is relatively small, the speedup is not notable, however, when the system scale is increased to 63 times of the original IEEE 39-bus system, the achieved speedup is up to 5.75.

Fig. 4.11 shows the execution time and speedup with increasing system size. It is obvious that the computation time of EMTP-RV follows a high-order complexity $O(n^a)$ $(a > 2)$ respecting to the system scale, since most vector and matrix operations have the high-order complexity, $O(n^2)$ and $O(n^3)$, in serial CPU algorithms. The execution time of the proposed MT-EMTP program, however, only increases linearly with first order complexity $O(n)$, derived from SIMD-based parallel programming. Thanks to the complexity order reduction, a GPU-based EMT simulator is always faster than the conventional CPU-based simulator when the scale of the test case is large enough. Therefore, the speedup can be expected to increase without saturation for increasing system sizes. Larger systems (greater than Scale 63) could also be tested on MT-EMTP, but the EMTP-RV licence only allowed a maximum

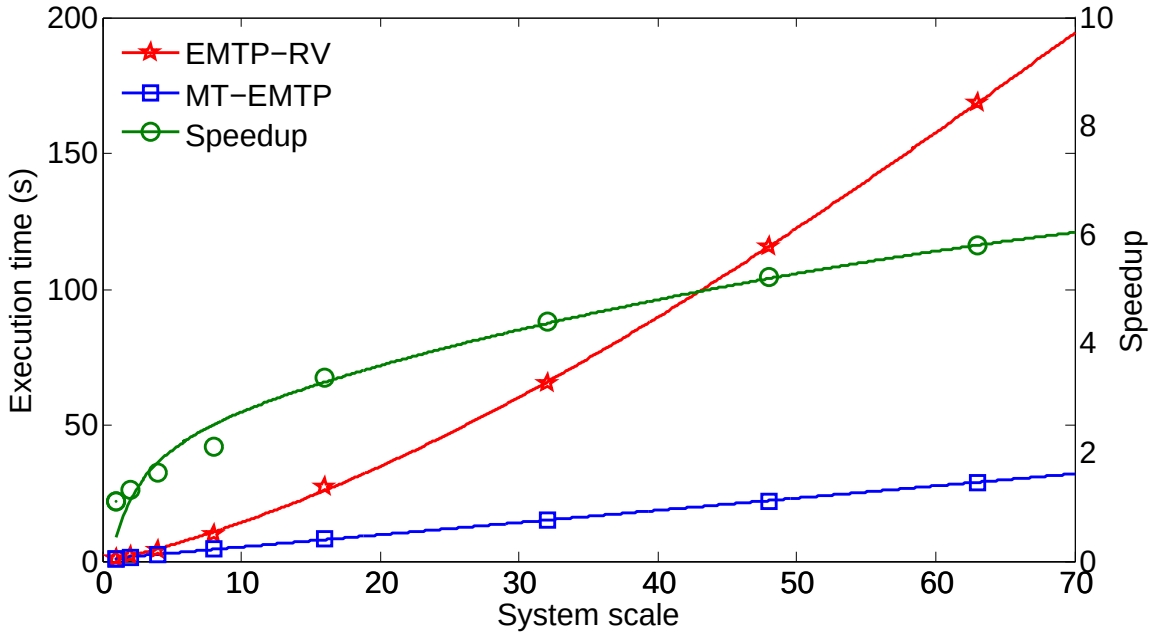

Figure 4.11: Execution time and speedup with respect to the scale of test systems in EMTP-RV and the GPU-based MT-EMTP program.

of 5000 devices. Note that in a commercial and industrial program like EMTP-RV there are many input and output activities, and a large collection of models/codes that require extra processing time. Nevertheless, this experiment clearly demonstrates the advantage of parallel massive-threading computation in accelerating EMT simulation.

4.2 Summary

In this chapter, the accuracy of the parallel massive-threading EMT program (MT-EMTP) was shown by the test cases for a fault on the transmission line and a fault on the machine; and the significant computational performance improvement is shown by the comparison test case between CPU and GPU EMT simulators. Although this parallel EMT simulator is still a laboratorial product, its performance in large-scale power system simulation and potential to reduce the computational complexity order are significant in comparison with the traditional commercial EMT simulator operating in series.

5 Conclusions and Future Work

Accuracy and speed are the eternal demands for the electromagnetic transient simulation of power systems in the modern electrical energy industry. Increasing the power system scale, the compromise has to be made between accuracy and speed since the limitation of the computational capability to traditional CPU-based EMT simulation system. The advent of GPU showing its extraordinary computational performance with the many-core structure and massive-threading architecture relieves this contradiction effectively. The power and potential of GPU-based massive-threading parallel EMT simulation shown in this thesis presents a new possibility to accelerate EMTP.

This thesis describes the development of a parallel massive-threading EMT program (MT-EMTP). Massive-threading parallel implementation of component models and numerical solvers used in power system for large-scale electromagnetic transient simulation is described. The proposed methods, algorithm and data structure can also be applied to a multi-threading computing system, which is also pervasive nowadays as a mainstream CPU architecture. The summary of the completed thesis work and directions for future work are presented in this chapter.

5.1 Contributions

- The models and methods developed in this simulation system cover the major application of EMT simulation. Using the developed modules, including ULPEM, ULM, UMM, NRIM and LU-FBSM, a complete power system, i.e., the IEEE 39-bus power system, can be simulated with linear passive elements, transmission lines, electrical machines and nonlinear elements.

- The transmission lines and electrical machines used in large-scale EMT simulation of

a power systems of up to 2458 buses, are modeled using detailed models, universal line model and universal machine model, which leads accuracy and generalization to the component models.

- Novel massive-threading parallel modules are developed for all components, including linear passive element, transmission line and machines, used in this parallel EMT simulator based on the GPU and CUDA.
- Novel massive-threading parallel modules are developed for numerical solvers, including Newton-Raphson iteration method and forward-backward substitution with LU factorization, for solving nonlinear and linear system used in this parallel EMT simulator based on the GPU and CUDA.
- As shown by the evaluations and comparisons with the commercial EMTP software using various large-scale test cases, the EMT simulation for large-scale power system is significantly accelerated by GPU-based massive-threading computation, which reduce the computational complexity order from high order ($O(n^a)$ $(a > 2)$) to linear ($O(n)$). Therefore, a stable increase of speedup is obtained without saturation growing with the power system scale.

5.2 Directions of Future Work

- Thanks to the modularized architecture of this parallel EMT simulation system, more new modules for new component models and numerical methods can be plugged in to extend the capability of MT-EMTP, such as transformer model, power electronics element model and conjugate gradient method.
- Since most of components in this EMT simulation system are uncontrolled, control system will be implemented in the next version of MT-EMTP. By modeling the cyber systems, it even can be used to simulate the transient phenomenon in a smart grid.
- To handle larger scale power system, the parallel EMT simulation system should be expanded from single GPU to multiple GPU computing system.

Bibliography

[1] H. W. Dommel and W. S. Meyer, "Computation of electromagnetic transients", *in Proc. IEEE,* vol. 62, no. 7, pp. 983-993, April 1974.

[2] CAN/AM EMTP Users Group, *Alternative Transients Program (ATP) Rule Book,* 2000.

[3] Manitoba HVDC Research Centre, *EMTDC User's Guide,* Winnipeg, MB, Canada, 2003.

[4] EMTP-RV website: www.emtp.com

[5] J. Mahseredjian, V. Dinavahi, J. A. Martinez, "Simulation tools for electromagnetic transients in power systems: overview and challenges", *IEEE Trans. on Power Delivery,* vol. 24, no. 3, pp. 1657-1669, July 2009.

[6] L. Gérin-Lajoie, J. Mahseredjian, "Simulation of an extra large network in EMTP: from electromagnetic to electromechanical transients", *Int. Conf. on Power System Transients,* IPST09-227, Kyoto, Japan, June 3-6, 2009.

[7] D. Luebke, G. Humphreys, "How GPUs work", *Computer,* vol. 40, no. 2, pp. 96-100, Feb. 2007.

[8] D. Blythe, "Rise of the graphics processor", *Proc. IEEE,* vol. 96, no. 5, pp. 761 - 778, May 2008.

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, "GPU computing", *Proc. IEEE,* vol. 96, no. 5, pp. 879 - 899, May 2008.

[10] W. Liu, B. Schmidt, G. Voss, W. Muller-Wittig, "Streaming algorithms for biological sequence alignment on GPUs," *IEEE Trans. on Parallel and Distrib. Syst.,* vol. 18, no. 9, pp. 1270-1281, Sept. 2007.

[11] R. Weber, A. Gothandaraman, R. J. Hinde, G. D. Peterson, "Comparing hardware accelerators in scientific applications: a case study," *IEEE Trans. on Parallel and Distributed Syst.,* vol. 22, no. 1, pp. 1045-9219, Jan. 2011.

[12] A. Gopal, D. Niebur, S. Venkatasubramanian, "DC power flow based contingency analysis using graphics processing units", *Power Tech, 2007 IEEE Lausanne,* pp. 731-736, 1-5 July 2007.

[13] J. E. Tate, T. J. Overbye, "Contouring for power systems using graphical processing units", *Proc. of 41st Ann. Int. Conf. on System Sciences, Hawaii, USA,* pp. 168, 7-10 Jan. 2008.

[14] V. Jalili-Marandi, V. Dinavahi, "Large-scale transient stability simulation on graphics processing units", *IEEE PES '09, Power & Energy Society General Meeting,* pp. 1-6, 26-30 July 2009.

[15] N. Garcia, "Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach", *IEEE PES General Meeting,* pp. 1-4, 25-29 July 2010.

[16] V. Jalili-Marandi and V. Dinavahi, "SIMD-Based large-scale transient stability simulation on the graphics processing unit", *IEEE Trans. on Power Systems, 2010,* vol. 25, no. 3, pp. 1589-1599, Aug. 2010.

[17] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit", *IEEE Ann. India Conf. (INDICON),* pp. 1-5, 17-19 Dec. 2010.

[18] C. Vilacha, J.C. Moreira, E. Miguez,and A.F. Otero, "Massive Jacobi power flow based on SIMD-processor", *10th Int. Conf. on Envir. and Elect. Engin. (EEEIC),* pp. 1-4, 8-11 May 2011.

[19] V. Jalili-Marandi, Z. Zhou, V. Dinavahi, "Large-scale transient stability simulation of electrical power systems on parallel GPUs", *IEEE Trans. on Parallel and Distributed Systems,* vol.23, no.7, pp.1255-1266, July 2012.

[20] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings,* pp. 483-485, 1967.

[21] Intel Corporation, "Granularity and Parallel Performance", *Intel Guide for Developing Multithreaded Applications,* 2010.

[22] D. Chen, H. Su and P. Yew, "The impact of synchronization and granularity on parallel systems", *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, Washington, USA,* 1990.

[23] NVIDIA Corporation, "NVIDIAs Next Generation CUDA™ Compute Architecture: Fermi™", 2009.

[24] NVIDIA Corp., *NVIDIA CUDA C Programming Guide Version 4.0,* 5/6/2011.

[25] NVIDIA Corporation, "Whitepaper NVIDIA GF100", 2010.

[26] Intel Corporation, "Intel® microprocessor export compliance metrics", 2011.

[27] H. W. Dommel, *EMTP theory book,* Bonneville Power Administration, 1984.

[28] J. R. Marti and J. Lin, "Suppression of numerical oscillations in the EMTP", *IEEE Trans. on Power Sys.,* vol. 4, No. 2, pp. 739-747, May 1989.

[29] H. W. Dommel, "Nonlinear and time-varying elements in digital simulation of electromagnetic transients", *IEEE Trans. on Power Apparatus and Systems,* vol. 90, no. 4, pp. 2561-2567, June 1971.

[30] H. W. Dommel, "Digital computer solution of electromagnetic transients in single and multiphase networks", *IEEE Trans. on Power Apparatus and Systems,* vol. PAS-88, no. 4, pp. 388-399, April 1969.

[31] A. Morched, B. Gustavsen, and M. Tartibi, "A universal model for accurate calculation of electromagnetic transients on overhead lines and underground cables", *IEEE Trans. on Power Delivery*, vol. 14, no. 3, pp. 1032-1038, July 1999.

[32] J. R. Marti, "Accurate modeling of frequency-dependent transmission lines in electromagnetic transient simulations", *IEEE Trans. on Power Apparatus and Systems,* vol. PAS-101, no. 1, pp. 147-155, January 1982.

[33] B. Gustavsen and A. Semlyen, "Simulation of transmission line transients using vector fitting and modal decomposition", *IEEE Trans. on Power Delivery*, vol. 13, no. 2, pp. 605-614, April 1998.

[34] H. K. Lauw and W. Scott Meyer, "Universal machine modeling for the representation of rotating electric machinery in an electromagnetic transients program", *IEEE Trans. on Power Apparatus and Systems,* vol. 101, no. 6, pp. 1342-1350, June 1982.

[35] H. K. Lauw, "Interfacing for universal multi-machine system modeling in an electromagnetic transients program", *IEEE Trans. on Power Apparatus and Systems,* vol. 104, no. 9, pp. 2367-2373, September 1985.

[36] L. Buatois, G. Caumon and B. Lévy, "Concurrent number cruncher: a GPU implementation of a general sparse linear solver", *International Journal of Parallel, Emergent and Distributed Systems,* vol. 24, no. 3, pp. 205-223, Jun. 2009.

[37] G. P. Krawezik and G. Poole, "Accelerating the ANSYS direct sparse solver with GPUs", *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC10),* 2010.

[38] R. F. Lucas, Gene Wagenbreth, Dan M. Davis, and Roger Grimes, "Multifrontal Computations on GPUs and Their Multi-core Hosts", *VECPAR 2010, LNCS,* vol. 6449, pp. 71-82, 2011.

[39] M. Garland, "Sparse matrix computations on manycore GPU's", *Proceedings of the 45th Annual Design Automation Conference ,* Anaheim, California, USA, pp. 2-6, Jun. 2008.

[40] R. J. Cichelli, "Minimal perfect hash functions made simple", *Comm. of the ACM,* vol. 23, No. 1, pp. 17-19, January 1980.

A

System Data of Case Study in Chapter 4

The parameters for the power system in Fig. 4.1 are given below:

1. ULM transmission line (Line1 - Line35) parameters: three conductors, resistance: 0.0583 /km, diameter: 3.105 cm, line length: 50km (line 5, 6, 7, 8, 15, 16, 18, 19, 23, 27, 29, 30, 31, 35), 150km (line 2, 3, 4, 9, 10, 11, 13, 14, 20, 21, 22, 24, 25, 26, 32, 33) and 500 km (line 1, 12, 17, 28, 34). $\boldsymbol{Y}$ and $\boldsymbol{H}$ are 3×3 matrices, whose elements are approximated with ninth-order rational functions (3.25) and (3.26). The geometry is shown in Fig. A.1.

2. UMM synchronous machine (G1 - G10) parameters: 1000 MVA, 22 kV, Y-connected, field current: 2494 A, 2 poles, 60 Hz, moment of inertia: 5.628e4 kg·m^2/rad and damping: 6.780e3 kg·m/s/rad. The winding resistances and leakage reactance (Ω) are listed in Table A.1.

3. Loads and transformer parameters: load parameter: R = 500Ω, L = 0.05H, C = 1μF and transformer leakage impedance: R = 0.5Ω, L = 0.03H.

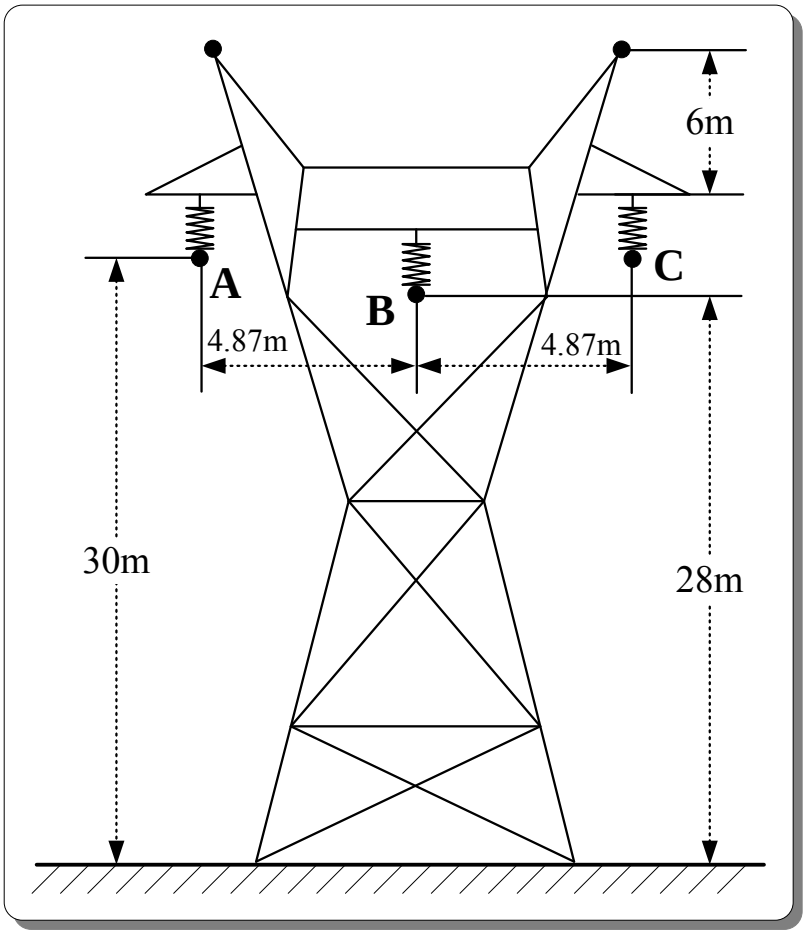

Figure A.1: Tower geometry of transmission lines in the case study

Table A.1: UMM machine parameters

R_d	9.680e-4	R_q	9.680e-4	R_0	9.680e-4
R_f	1.111	R_{D_1}	3.499	R_{D_2}	5.571
R_{Q_1}	7.627e-1	R_{Q_2}	1.227	R_{Q_3}	2.096e2
X_d	6.747e-1	X_q	6.549e-1	X_0	9.099e-2
X_f	2.392e2	X_{D_1}	2.067e2	X_{D_2}	5.571
X_{df}	8.821	X_{dD_1}	8.821	X_{dD_2}	8.821
$X_{D_1D_2}$	2.066e2	X_{fD_1}	2.066e2	X_{fD_2}	2.099e2
X_{Q_1}	4.453e2	X_{Q_2}	2.218e2	X_{Q_3}	2.096e2
X_{qQ_1}	8.521	X_{qQ_2}	8.521	X_{qQ_3}	8.521
$X_{Q_2Q_3}$	1.577e2	$X_{Q_1Q_2}$	1.577e2	$X_{Q_1Q_3}$	1.577e2

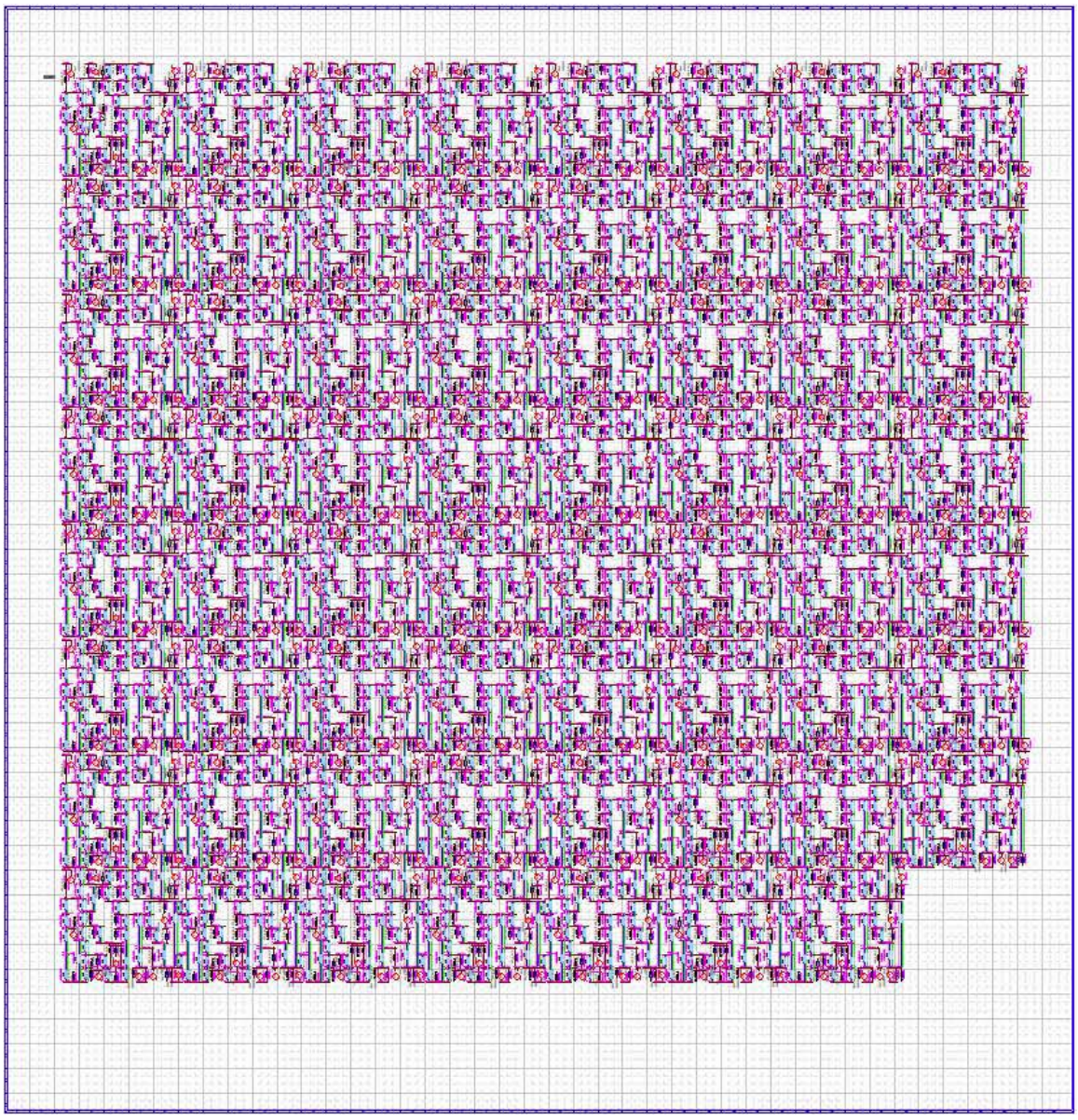

Figure A.2: The snapshot of the Scale 63 large-scale power system in the EMTP-RV® software.