

國立交通大學

電子工程學系 電子研究所

碩士論文

三維積體電路在通用圖形處理器裡基於
力使量法的平行分割演算法



**A Force-Directed Based Parallel Partitioning
Algorithm for Three Dimensional Integrated
Circuits on GPGPU**

研究生：陳琬菁

指導教授：賴伯承 教授

中華民國 一〇〇 年 八 月

三維積體電路在通用圖形處理器裡基於

力使量法的平行分割演算法

**A Force-Directed Based Parallel Partitioning
Algorithm for Three Dimensional Integrated
Circuits on GPGPU**

研究生：陳琬菁

Student : Wan-Jing Chen

指導教授：賴伯承

Advisor : Bo-Cheng Lai



Submitted to Department of Electronics Engineering and
Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

In partial Fulfillment of the Requirements

For the Degree of

Master

In

Electronics Engineering

August 2010

Hsinchu, Taiwan, Republic of China

中華民國 一〇〇年 八月

三維積體電路在通用圖形處理器裡基於 力使量法的平行分割演算法

研究生：陳琬菁

指導教授：賴伯承教授

國立交通大學

電子工程學系 電子研究所 碩士班

摘要

本論文提出一個創新的平行演算法(稱為FDPrior)，利用力使量法(force-directed)解決在3DIC中的多層次分割問題(multilayer partitioning problem)，我們的研究主要提供一個全新的角度去思考如何解決分割問題；由於3DIC技術中層次架構及規模日益擴大，需要昂貴的計算過程才能達到優化目的，利用多核心架構的平行性將成為關鍵，並可縮短運行時間，我們研究目標是盡量減少TSV的總數量，且同時滿足每一層晶片的面積限制。藉由N-body simulation方案及新技術達到減少不必要的同步次數，FDPrior成功地在通用圖形處理器(GPGPU)架構裡開發出大量平行度；使用ISPD98當作輸入並做實驗測試，FDPrior平均上比傳統FM演算法獲得5.95倍更好的實驗結果，並加速高達303.66倍的運行時間；而跟PP3D相比，FDPrior平均仍然可達到7.71倍更好的結果，和增強3.35倍的運行時間。

近年來，多階層超圖(multilevel hypergraph)分割演算法比非多階層的方法可得到更好性能。因此，該論文也提出一個新的演算法稱為MFDPrior，它是使用之前所提的FDPrior當作基本分割演算法，採用多階層演算法作為骨架，跟之前所提的FDPrior演算法相比，MFDPrior平均可獲取1.46倍更好的實驗結果，和贏得1.44倍的時間加速。

A Force-Directed Based Parallel Partitioning Algorithm for Three Dimensional Integrated Circuits on GPGPU

Student: Wan-Jing Chen

Advisor: Bo-Cheng Lai

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

ABSTRACT

This thesis proposes an innovative force-directed parallel algorithm, FDPrior, to solve the multilayer partitioning problem of 3DICs. The purpose of our research is providing a new field of vision in the partition problem of 3DICs. The growing scale and multi-layered structure of the 3DIC technology make it computationally expensive for EDA tools to achieve optimization goals. Exploiting the algorithmic parallelism on multi-core architectures becomes the key to attain scalable runtime. The objective is to minimize the total number of Through Silicon Vias (TSVs) while meeting the area constraint for each layer. By adopting the N-body simulation scheme and novel techniques to reduce synchronization overhead, FDPrior successfully exposes the massive parallelism on the multi-core GPGPU architecture. The experimental results on ISPD98 benchmark show that FDPrior outperforms the conventional FM algorithm by achieving in average 5.95X better TSVs and up to 303.66X runtime speedup. Compared with PP3D, a parallel 3DIC partitioning algorithm, FDPrior achieves 7.71X better TSVs with 3.35 X runtime enhancements.

In recent years, the multilevel hypergraph partitioning algorithms could earn better performances than non-multilevel methods. This is why our thesis also proposes an algorithm, MFDPrior, which fulfills the multilevel framework. MFDPrior exercises the FDPrior as the essential partitioning part. When comparing with the single level FDPrior, MFDPrior demonstrates an average of 1.46X better solution quality and earns 1.44X speedup.

致謝

本篇論文得以完成，首先要感謝指導教授賴伯承博士的辛苦栽培，在兩年的過程中，教導我研究的態度和方法，並且適時地給予建議和方向。也感謝實驗室的同學、學長、學弟妹們的鼓勵和幫助，使得研究能夠順利進行，並度過了快樂的兩年研究生的生活。最後，也要謝謝一直在背後默默支持我的家人和朋友們，特別感謝蘇俊仁、林佩潔、吳木陳、楊子嫻、游雅蘭給我的鼓勵。由衷地感謝大家。



民國一百年八月
研究生陳琬菁謹識於交通大學

Contents

Chinese Abstract	I
English Abstract	II
Acknowledgement	III
Contents	IV
List of Figures	VIII
List of Symbols	X
Chapter 1 Introduction	1
Chapter 2 Introduction of N-body Problem	3
2.1 Definition of N-body Problem	3
2.2 Difference Formation of Newton N-body Problem.....	5
2.3 General Considerations: Solving the N-body problem	6
Chapter 3 N-body Simulation on GPU	7
3.1 The GPGPU Parallel Platform	7
3.1.1 CUDA	8
3.1.2 Comparison between A GPU and A CPU	9
3.2 Design Methodology	10
3.3 Design Considerations On GPGPUs	11

3.3.1	The Amount of Data Used by Threads	11
3.3.2	The Determined Number of Threads	12
3.4	Optimization Methods On GPGPUs	13
3.4.1	Coalescing Access	13
3.4.2	Reduction Technology	14
3.5	Consideration of Memory Capacity	16
3.6	Inter-cell Force Modeling	18
Chapter 4	Case Study: 3D ICs Partitioning	20
4.1	Introduction Of the 3DIC Partitioning	20
4.2	Related Work: Partitioning Algorithms	21
4.3	3DIC Partitioning Problem Formulation	22
4.3.1	The Structure of 3DICs	23
4.3.2	Variable Definitions	23
4.3.3	Problem Description	24
Chapter 5	FDPrior Algorithm	25
5.1	FDPrior algorithm.....	26
5.1.1	Phase1: N-body Simulation	27
5.1.2	Phase 2: Mapping Cells To A Layer	31
5.1.3	Phase 3: Escape From Local Optimum	32
5.2	Experimental Results	33
5.3	The Discussion of FDPrior	36
5.3.1	Comparisons Between algorithms	37
5.3.2	The Distribution Of Layer Spaces	37

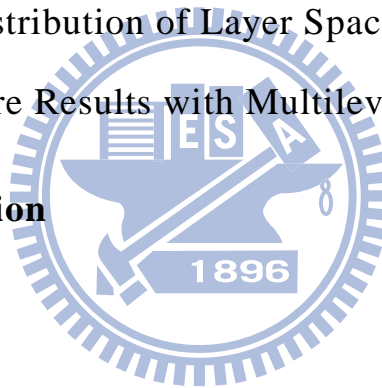
5.3.3	Parallel And Sequential Scemes On N-body Simualtion	39
-------	---	----

Chapter 6	MFDPrior: Multilevel Of FDPrior	41
------------------	--	-----------

6.1	Introduction of Multilevel	41
6.2	MFDPrior: The Multilevel Methodology of FDPrior.....	42
6.2.1	Multilevel Coarsening Phase	44
6.2.2	Partitioning Phase	45
6.2.3	Multilevel Un-coarsening Phase	45
6.3	Experimental Results	46
6.4	Discussion of Issues	48
6.4.1	The Distribution of Layer Spaces	49
6.4.2	Compare Results with Multilevel of PP3D	49

Chapter 7	Conclusion	51
------------------	-------------------	-----------

Bibliography		52
---------------------	--	-----------



List of Tables

Table 1: The reduction technique in a warp	15
Table 2: Sums of data by using reduction technique	16
Table 3: The code of the calculated forces in the N-body simulation phase	30
Table 4: Characteristics in ISPD98 benchmark	33
Table 5: The average number of TSVs execute on the algorithms in each case.	35
Table 6: Comparison of runtime on the algorithms	36
Table 7: The distribution of all layer areas of FDPrior in ISPD98 benchmark	38
Table 8: Comparisons of average solutions and runtime with FDPrior and MFDPrior	47
Table 9: The distribution of layer areas of MFDPrior in ISPD98 benchmark	48

List of Figures

Figure 1: The architecture of Nvidia’s Geforce 9800GT.	8
Figure 2: Parallel design methodology which guides the design optimization from algorithm to the parallel platform.....	10
Figure 3: Memory segments and thread in a half warp of thread.....	12
Figure 4: Two arrays that are used to describe in Fig.9-1.....	14
Figure 5: The simple parallel reduction technology.....	14
Figure 6-1: Definition and initial hypergraph.....	17
Figure 6-2: The total amount of memory of each module is smaller than 1GB.....	17
Figure 6-3: Module A is larger than 1GB.....	17
Figure 7-1: Multi-pin nets.....	18
Figure 7-2: Clique models for Fig.7-1.....	18
Figure 8-1: A hypergraph $G=(C, Net)$ and its initial solution.....	19
Figure 8-2: The given hypergraph translates into clique model.....	19
Figure 8-3: Cell B is affected by the set S_B and $n_B=5$	19
Figure 9: The structure of a 3DIC for vertical interconnects.	23
Figure 10: The flow chart of FDPrior algorithm.	26
Figure 11: The force by a stretched spring joining cell i and j.....	29
Figure 12: Comparison of average solutions on the algorithms.	34

Figure 13: The distribution of solution qualities of algorithms in ibm04 35

Figure 14: Comparisons of the average solutions between parallel and sequential schemes of N-body simulation 39

Figure 15: The simple phases of the multilevel partition algorithm 42

Figure 16: The flow chart of MFDPrior algorithm 43

Figure 17: Simple diagram of modified hyperedge coarsening 44

Figure 18: Comparisons of average number of TSVs with FDPrior and MFDPrior .. 46

Figure 19: Comparison of average TSVs with Multilevel of PP3D 50



List of Symbols

G	The given hypergraph of a netlist
C	A set of cells that each cell $c_i \in C$.
N	The size of C or the total number of particles in a N -body system
Net	A set of nets that connect to two or more cells
K	Divides the set of cells C into K layers ($K \geq 2$)
F_{ub}	A given constant that decides the range of area bounds
A_{layer_j}	The total area of all cells in layer_j
A_{IO}	The total area of the TSV_IO cells
A_{avg}	The average layer area of a 3DIC
A_{cell_i}	The given area of cell c_i
A_{min}	Minimum bound of a layer space
A_{max}	Maximum bound of a layer space
S_i	A set of adjacent cells and pins to cell c_i
n_i	The total number of elements in the set S_i .
n_{avg}	The average of the summation of n_i ($=\sum n_i/\text{size of } C$)
Δx_i	The displacement of c_i between two iterations(in z-direction)
F_i^{hold}	The hold force impacts on cell c_i
$F_i^{attract}$	The attractive force impacted on a cell c_i .
F_i^{total}	The total force of s cell c_i
<i>Number of TSVs</i>	Contains both the TSV_IOs and TSVs.
σ	The standard deviation of all layer areas in the circuit
CV	The layer area coefficient of variation
k	The spring constant of a spring

Chapter 1

Introduction

A Graphics Processing Unit (GPU) is a multi-processor which is used to offload intensive computations. Authentic speedups can be achieved by controlling the power on certain compute-intensive applications. The advantages of GPU technologies have propelled the GPU into a sea change in computer architecture due to the impending ubiquity of multi-processors. However, programmer's traditional algorithms and the selection of algorithms being used for problems require changes to utilize GPUs [1]. This paper adopts a highly parallel N-body simulation scheme on EDA problems and maps on a GPGPU to benefit from its massive parallel computation capability.

N-body simulation can describe the interaction of N particles in a system. Each of N particles affects others according to a function of their separation distances [2]. In computational physics, N-body algorithms are usually relative to many common problems in gravitation, fluid dynamics and electrostatics. The all-pairs direct approach to N-body simulation is a straightforward method that sum of all pair-wise interactions among the N particles. This naïve method requires $O(N^2)$ computational complexity, which is clearly not fast enough in the simulation of large systems. Parallel computation is considered as a good solution to speed calculations. Nvidia had researched in N-body problem and discovered that all-pairs N-body algorithm is special suited to execute on GPU platforms [3].

Our research provides three contributions. The first important contribution is the translation from exploring electronic design automation (EDA) field into applications of N-body algorithm [4]. Besides, our research attempts to transfer interaction model

from traditional all-pairs into partial-pairs, which is more proper to electric circuits and real-world algorithms.

The second strength is to reduce the number of synchronizations by our proposed algorithm, FDPrior. Three-dimensional integrated circuits (3DICs) technology has been considered as a solution to the challenges of large die area and long global wire delay in the advanced semiconductor technology [5]. Accordingly, we deliberate a study case in the partitioning problem on 3DICs by N-body algorithm. Under these considerations, we proposed a novel multi-layer partitioning algorithm on GPU platforms, which is called FDPrior. FDPrior can efficiently reduce the unnecessary synchronizations by implying bottom-up layer constructions.

The third contribution is to join a multilevel structure on FDPrior and improve the solution qualities. Multilevel approach is one of famous algorithms to solve partitioning problems. Multilevel method can coarsen the size of original hypergraph and solve each level hierarchically. Because of the enhancement of solutions by multilevel methods, we also implements multilevel methodologies on the proposed FDPrior algorithm, and called this modification algorithm as MFDPrior.

The rest of this thesis is organized as follows. Chapter 2 introduces the traditional N-body simulation in detail. Chapter 3 provides the GPU platform architecture, detailed optimization steps and parallel design methodology on a GPU. Chapter 4 presents the related work on partitioning problems and problem formulation. The overview of the proposed FDPrior algorithm flow, the experiment results on ISPD98 benchmark and probably issues about FDPrior are discussed in Chapter 5. Chapter 6 focuses on the multilevel approach of FDPrior algorithm, MFDPrior. Besides, the discussion of MFDPrior and experimental results also are provided in Chapter 6. Finally, the conclusions are drawn in Chapter 7.

Chapter 2

Introduction of N-body Problem

N-body simulation is a simulation of N particles in a dynamical system, and is usually under the influence of physical forces, such as gravity. In Chapter 2, we introduce some basic concepts for helping to understand the fundamental of N-body problems.

2.1 Definition of N-body Problem

Formally, N-body problem is a problem which describes the motion of N particles that interacts with others [6]. Each particle affects all others according to a function of their separation distance. The informal version of the N-body problem is described as following [7]:

In a dynamical system, there have N particles in space which masses are $m_1 \dots m_N$. In the beginning, only the present conditions and initial positions for every particle are specified at the current instant. Then determine the position of each particle for the future time or even past time in this system. In mathematical terms, the N-body problem describes the process of finding a global solution of the initial value problem.

N-body algorithms have numerous applications such as astrophysics, molecular dynamics and plasma physics. The all-pairs method is a brute-force approach that evaluates all pair-wise interactions among the N particles. The direct method is a

naïve solution to solve N-body problem, which sums the individually-computed forces on a given particle over all the mutual interactions [8]. However, this all-pairs directed technique requires $O(N^2)$ computational effort for the interaction of N particles among a system. Due to the required intensive mathematical computations, N-body problem encounters a formidable challenge to the numerical analysis and computer hardware.

The force of attraction experienced between each pair of particles is a constant Newtonian force. In the view of physics, N-body problem determines the motion of N particles attracting one another in pairs according to the Newton law of gravity under a given initial condition which contains their positions and velocities. In the current popular models, the interactions between particles essentially concentrate only on their self-gravity. The N-body simulation provides an evolution of a self-gravitating finite system. In the view of a self-gravitating system, each particle feels the force attraction of all the other particles and interests primarily in the dynamics system which is developed from initial states [9]. The lack of anti-gravity or reaction is the mainly arduous issue in simulating systems.

We believe that the equations of motion are not purely gravity in N-body problem. If the definitions of force attraction between each pair of particles are applicable, the problems could be handled properly. The numerous applications in N-body problems are not only on physics but also on electronic design automation (EDA). We are primarily interested in the modified equations of the interactions, as opposed to the original Newton's Law of Gravity. In this thesis, our research presented an approach to provide an insight about the nature of existing approximations to self-action systems instead of self-gravitating systems.

2.2 Difference Formation of Newton N-body Problem

Isaac Newton's monograph was published in 1687 and comprised the foundations for most of classical mechanics. Since gravity is responsible for the motion of stars and planets, Newton is among the first who completed the mathematical formulation which presents gravitational interactions in terms of differential equations in a system.

In a system which contains N particles, the equations of motion for a particle of index i can be described in the following form:

$$\ddot{r}_i = -G \sum_{j=1, j \neq i}^N \frac{m_j (r_i - r_j)}{|r_i - r_j|^3} \quad (1)$$

Equation 1 is Newton's second law of motion. For a particle of index j , m_j is the mass and r_j is the coordinates. For convenience, the gravitational constant G usually scales to one unit (that means G is equal to one). The power in the denominator is three instead of two to balance the vector difference and be used to specify the direction of the force.

Equation 2 is a modified expression including a softening parameter (ϵ) [10].

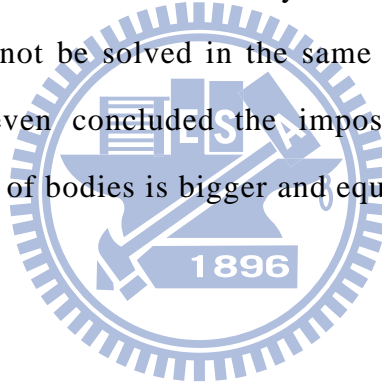
$$\ddot{r}_i = -G \sum_{j=1, j \neq i}^N \frac{m_j (r_i - r_j)}{(|r_i - r_j|^2 + \epsilon^2)^{3/2}} \quad (2)$$

The softening parameter is introduced in the modified equation on account of preventing the force singularity as the separation distance $r_{ij} \rightarrow 0$. Informally, a self-gravitating system is called collision-less system if and only if the mass distribution does not influence its evaluation. However, pure Newtonian interactions listed in Equation 1 ($\epsilon = 0$) are used for most applications.

2.3 General Considerations: Solving the N-body problem

Newton describes the three laws of motion and the law of gravity. These Principia is generally considered to be one of the greatest scientific accomplishments of all times. In the N-body problem, N defines the number of bodies in a system. With these principles, Newton completely demonstrated that the two-body ($N = 2$) problem deriving Kepler's laws of planetary motion and his theory of gravitation.

The two-body problem could be completely solved by Johann Bernoulli [11]. In the case of $N = 3$ (called three-body problem), strict solutions only exist in some special cases. The N-body problem only confesses the exact solutions in the case of two interacting particles. It has been widely known that the N-body problem in Equation 1 for $N \geq 3$ cannot be solved in the same sense as the two-body problem. Some physic literature even concluded the impossibility of solving the N-body problem when the number of bodies is bigger and equal than three ($N \geq 3$) [12].



Chapter 3

N-body Simulation on GPUs

Chapter 3 mainly illustrates how N-body simulation is implemented on a GPGPU. Since our experiments are implemented on this platform, GPGPU architecture of Nvidia Geforce 9800GT is provided below. Chapter 3 presents the parallel design methodology which guides optimizations from a parallel algorithm to a parallel architecture. And following sections discuss design considerations on GPUs, required optimization steps, and translation of hypergraph for collecting mutual forces in N-body simulation. Besides, memory restrictions on multi-core systems are discussed.

3.1 The GPGPU Parallel Platform

Our algorithms are implemented on the Nvidia Geforce 9800GT and the architecture is shown in Fig 1. The 9800GT includes 64 streaming processors (SP) which is grouped into 8 streaming multiprocessors (SM). Instructions will be fetched and decoded by SM and executed by eight SPs in the SM. In addition, each multiprocessor also has 8192 registers, 16KB share memory, the texture cache and constant cache.

The following section will introduce about GPU platform in detail. Section 3.1.1 introduces CUDA (Compute Unified Device Architecture) structure. CUDA is Nvidia's parallel platform which provides several APIs to make parallel programming easier. In CUDA, most essential operations are executed by SPs. And Section 3.1.2 enumerates GPU's benefits.

3.1.1 CUDA

CUDA is mainly based on the C-language which can let programmers quickly learn programming language. In CUDA structure, the region of executable program is dividing into two parts: Host and Device. In substance, host refers to the CPU side and device is the GPU platform. Usually the program will prepare ready information on the host and copy these data into memory of the GPU, and the GPU executes calculations on the device. Then the program on the host accesses completely data back from memory of the GPU. This memory translation takes extremely long latency as a result of only passing through PCI Express interface on a CPU. The above memory translation cannot be accessed too often for avoiding reducing efficiency.

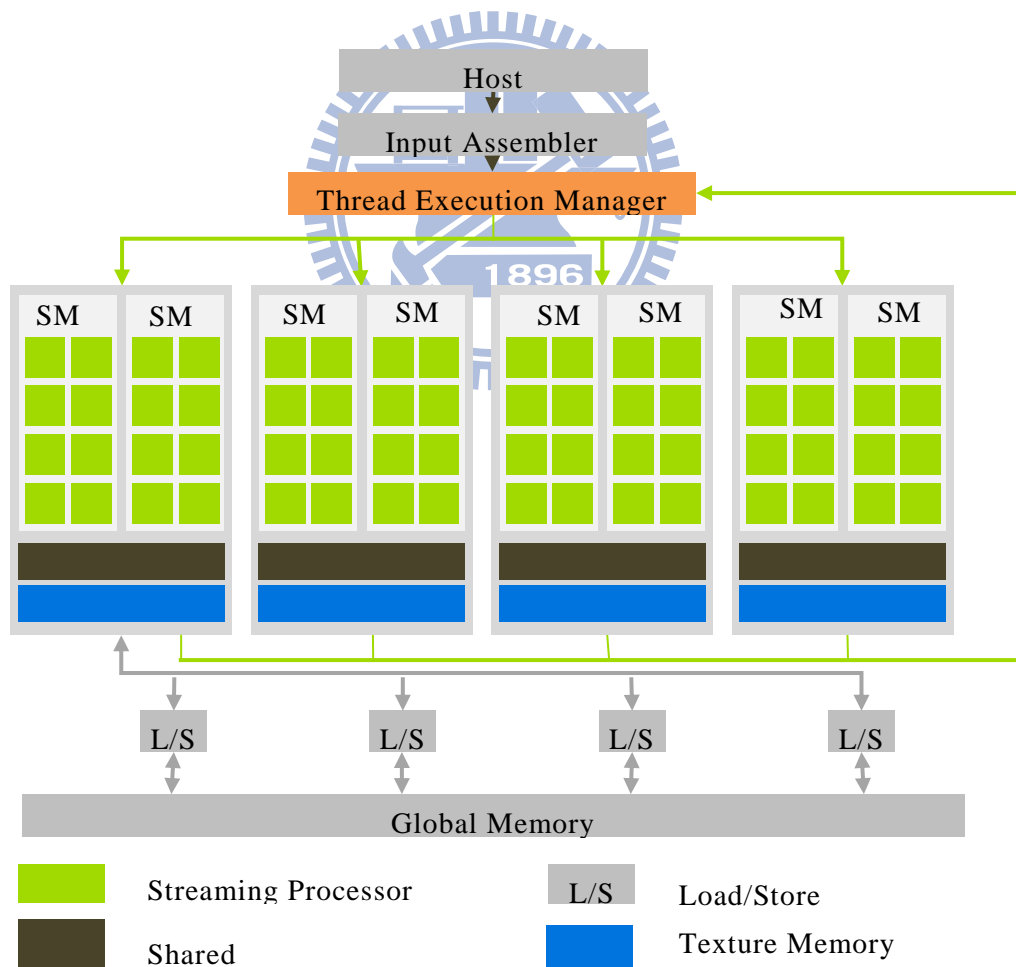


Figure 1: The architecture of Nvidia's Geforce 9800GT.

CUDA platform uses SIMT (Single Instruction Multiple Thread) technique to exploit the parallelism of a program. In a SIMT program, every thread executes the same instructions on different data regions. In the CUDA architecture, minimum execution unit on a GPU is a thread. Many threads are grouped into a thread block, and thread blocks are grouped into a grid. Each thread has own space of registers and local memory. All threads in a thread block are executed by a SM, and share the same resources within the SM. But threads in different thread blocks cannot access the same shared memory and cannot communicate directly or synchronization. Therefore, the extent of cooperation of threads is relatively low with different thread blocks. The CUPA program is based on the units of warp in implementation. A warp currently has 32 threads which divided into two groups of 16 threads (half-warp).

3.1.2 Comparison between A GPU and A CPU

Compared with CPUs, using GPUs has a few major benefits to operate works. Above all, a GPU has numerous execution units but lower clock rate. On the contrary, a CPU usually has less execution units but higher clock rate. Since the numerous execution units on a GPU, a GPU cannot bring much help for works with low degree of parallel. Besides, a GPU normally does not own complex flow control units, efficiency will be relatively poor when draw on high degree of branching programs. And a GPU usually possesses more memory bandwidth, such as the memory bandwidth on Nvidia's Geforce 9800GT is around 57 (GB/sec) and the currently high-class CPU just has around 10 (GB/sec). Moreover, the price of a GPU is cheaper than the price of a high-class CPU. However, current GPU programming model is still not mature and not yet recognized standards. Overall, a GPU platform is similar to a stream processor and is suitable to conduct a great deal of the same works. A CPU is more flexible which can conduct more various works simultaneously.

3.2 Design Methodology

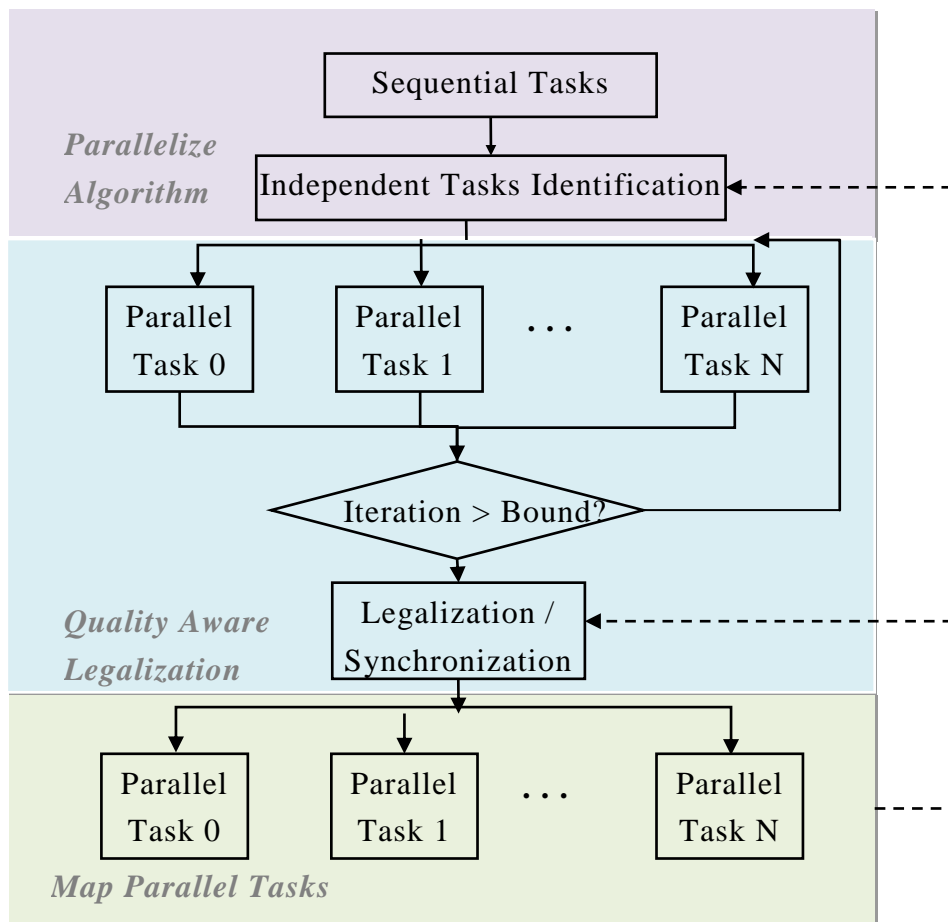


Figure 2: Parallel design methodology which guides the design optimization from algorithm to the parallel platform

For achieving faster runtime, an appropriately co-optimized technique is demanded to handle the communications between the parallelized algorithm and the characteristics of the parallel execution platform. The parallel design methodology, which widely used to algorithms in this thesis, is introduced in this section. In spite of our case study only focuses on 3DIC partitioning on GPGPU, we believe that this design methodology can be applied for other algorithms as well as different parallel architectures.

Fig.2 shows the flow of the parallel design methodology. This parallel design methodology can be divided into three fundamental phases. At first, the parallelize algorithm phase identifies a set of independent parallel tasks from the original

sequential algorithm. This first phase can expose the latent parallelism which could assist with speeding up the runtime of the architecture. Secondly, the quality aware legalization part can relax the synchronization criteria among parallel tasks within a certain execution period. The relaxed synchronization criteria allow the parallel tasks to search even larger problem space for high quality results. Afterwards, specific mechanisms are added to coordinate concurrent execution of the parallel tasks without violating constraints while considering the quality of the solution. Finally, the last part performs a seamless mapping of parallel tasks to the underlying parallel execution platform. In the last part, taking advantages of the useful features of parallel platform and avoid architectural bottlenecks is the major work. For example, memory conflicts, which are caused by massive communication among parallel tasks, could be the common bottleneck without careful designs. The dashed lines on the right side of Fig.2 illustrated the circumstances that several iterations usually are required between exposing the parallelism and appropriate optimizations in the parallel platform. Based on this design methodology, the following chapters discuss about how algorithms are designed as well as optimized for GPGPU in detail.

3.3 Design Considerations On GPGPUs

Section 3.3.1 clarifies coalescing of a half warp of threads, and discusses a specified amount of data which are loaded from threads. In Section 3.3.2, the number of threads per block and the number of blocks per grid are determined.

3.3.1 The Amount of Data Used by Threads

In the present CUDA architecture, coalescing global memory accesses is one of the most important performance considerations in programming. Coalescing access is quite useful to possess some local coherence in the texture. When definite access

requirements are met, global memory loads and stores by threads of a half warp (or a warp) are coalesced into few transactions by the device [13]. Global memory should be aligned segments of 16- and 32- words to confirm these access requirements: for instance, if every thread all access 32-bits data, then the address which is accessed by the first thread must is a multiple of 64-bytes (16*4 bytes). Fig. 3 explains coalescing of a half warp of threads, such as floats are 32-bit words, and shows global memory as rows of 64-byte aligned segments.

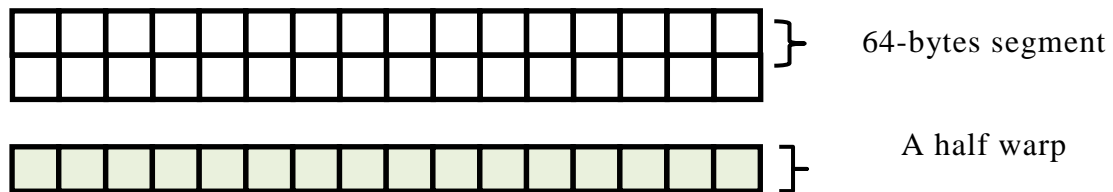


Figure 3: Memory segments and thread in a half warp of thread

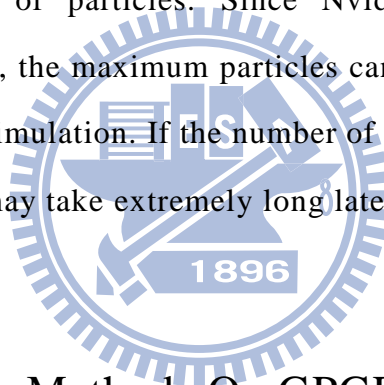
In current CUDA devices, an amount of data which is loaded from every thread can be 32-bites, 64- bits or 128-bits. If this amount of data does not meet the requirements, you can use the `__align(n)` instruction to solve problems. However, using 32-bits is the best efficiency. The used data types in our algorithms are integer or float types which all are 32-bit words. Because a GPU normally support 32-bits float type and probably not fully support IEEE 754 specification, some operations may be less accuracy.

3.3.2 The Determined Number of Threads

In CUDA thread hierarchy, threads are grouped into a thread block, and thread blocks are grouped into a grid. The number of threads per block and the number of blocks per grid may affect the performance. How to determine these numbers is depended on algorithms. The number of threads per block multiplied by the number of blocks per grid is the totally number of threads that executes on a GPGPU.

Each multi-processor has 8192 registers in the current CUDA device. If each thread uses 32 registers, a multi-processor only can maintain up to executions of 256 threads simultaneously. If the numbers of present threads more than this figure, it will reduce the efficiency of implementation when a part of data must be accessed in global memory. The maximum threads per blocks of the prevalent CUDA device are 512. For safety, the fixed number of threads per block on proposed programs is 256 when considering the limitation of registers.

Owing to parallel whole particles in N-body simulation, the numbers of blocks per grid is decided the number of particles divide by 256 which presents the number of threads per block. The number of threads that executes on the GPGPU must be larger than the number of particles. Since Nvidia Geforce 9800GT furnishes maximum 65535 grid size, the maximum particles can reach around 16 million which is big enough in N-body simulation. If the number of particles is more than 16 million, the execution of threads may take extremely long latency and cause worst efficiency.



3.4 Optimization Methods On GPGPUs

Although GPGPU provides highly parallel computational capability, without careful designs, the architectural bottleneck can easily limit the potential performance enhancement. Our programs apply two techniques to enhance performance. The first one is coalescing access to alleviate memory bottleneck and the second one is the parallel reduction to speed up the synchronization among all cells. These common optimized topologies are also suitable to other wide-ranging applications.

3.4.1 Coalescing Access

If every multiprocessors have belong global memory caches in the GPU platform, it will need cache coherence protocols and substantially raise the complexity of

cache. Since multiprocessors do not do cache in the global memory, the accessed latencies in the global memory are very long. Because of the accessed characteristics of the DRAM, the access in global memory is as continuous as possible.

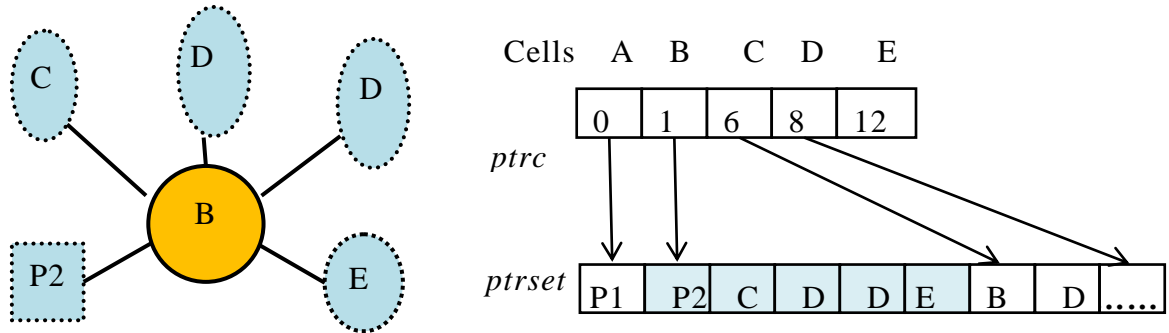


Figure 4: Two arrays that are used to describe in Fig.9-1.

For continuous access, cells are placed into an array where the addresses are continuous. As shown in Fig.4, we use two arrays to describe this data construction. The array, *ptrc*, is used to index *ptrset* array that stores the set of adjacent cells. Each cell is stored as a sequence of the cells that it spans, in consecutive locations in *ptrset*. When accessing all neighbors within a thread block, this data structure qualifies the coalescing access. For shorten the memory access latency, all threads in a block coalescing access and use the shared memory as a manual controlled cache.

3.4.2 Reduction Technology

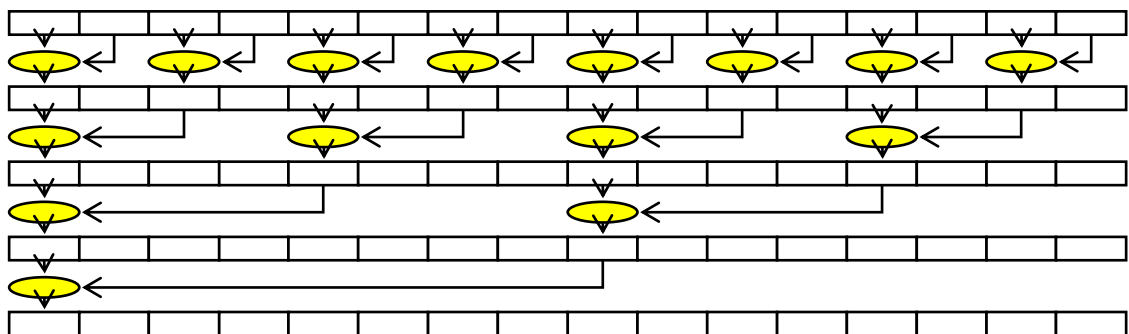


Figure 5: The simple parallel reduction technology

Even though we reduce the number of unnecessary synchronizations, the algorithm still requires some synchronization to collect and calculate information among cells. Therefore we use the parallel reduction technique [14] to speedup these evaluations. As shown in Fig.5, the computation complexity of an array with size of n could be reduced from $O(n)$ to $O(\log_2 n)$ in the program.

```

1  __global__ void  Devaluate_TSVs(){
2      __shared__ int sdata[256];           //declare shared memory
3      int tid = threadIdx.x;               //threads id in a block
4      int i = blockIdx.x*256+ tid;
5      sdata[tid] = 0;                     //initial shared memory
6      //each thread loads one element from global to shared memory
7      if(i<net_size)sdata[tid] = dnet_highlow[i];
8      __syncthreads();
9      if (tid<128) sdata[tid] += sdata[tid + 128]; //do reduction in shared memory
10     __syncthreads();
11     if (tid < 64) sdata[tid] += sdata[tid + 64];
12     __syncthreads();
13     if (tid < 32) warpReduce(sdata, tid); //reduction in a warp
14     //store result for this block to global memory
15     if (tid == 0) dodata[blockIdx.x] = sdata[0];
16 }

```

Table 1: The reduction technique in a warp

Table 1 illustrates code of the reduction technique in our following programs. This code sums of data by threads. Using `__shared__` instruction accesses the shared memory which access speed is quite fast, listed in line 2. In CUDA devices, `__syncthreads()` instruction, shown in line 8, is a built-in function and denotes that all threads of a block must be synchronized to this point. When the number of needful calculations is less than 32, we have only one warp left. We don't need to

__syncthreads() for less 32 threads per block and saves lot of time. Hence, we construct additional code to handle part in Table 1 line 13. Table 2 demonstrates the reduction technique in a warp.

```

1  __device__ void  warpReduce ( volatile int *sdata, const int tid ) {
2      sdata[tid] += sdata[tid + 32];
3      sdata[tid] += sdata[tid + 16];
4      sdata[tid] += sdata[tid + 8];
5      sdata[tid] += sdata[tid + 4];
6      sdata[tid] += sdata[tid + 2];
7      sdata[tid] += sdata[tid + 1];
8  }
```

Table 2: Sums of data by using reduction technique

3.5 Consideration of Memory Capacity

The total amount of global memory available on the device is almost 1GB in Nvidia Geforce 9800GT. Even though the amount of available memory is big enough for our use in ISPD98 benchmark, the total amount of memory in GPU is still smaller than in CPU. We probe into memory problems when the amount of global memory is not big enough for applications. Because implementations on GPU are only for the N-body simulation of our proposed algorithms, we just discuss the configuration of memory in N-body simulation part.

Finding out independent modules of a hypergraph is the first step. If the memory size of each independent module in a circuit is smaller than the maximum amount of global memory on a GPU, the order and association of the independent module can be easily arranged in the usable memory. These independent modules can practice individually in N-body simulation. Since these modules are self-reliant, there will not have any impact on forces between different independent modules and do not cause

influence on solutions. For example in Fig. 6, a given hypergraph is composed by three independent modules: module A, B and C. The total amount of required memory of each module is smaller than 1GB. At first, program performs movements of module A until completely exercising in N-body simulation phase. Then, the program replaces the required second data between CPU and GPU, as well as module B and C are accomplished finally.

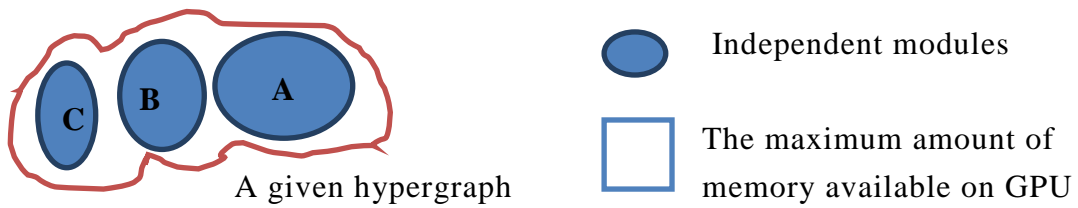


Figure 6-1: Definition and initial hypergraph

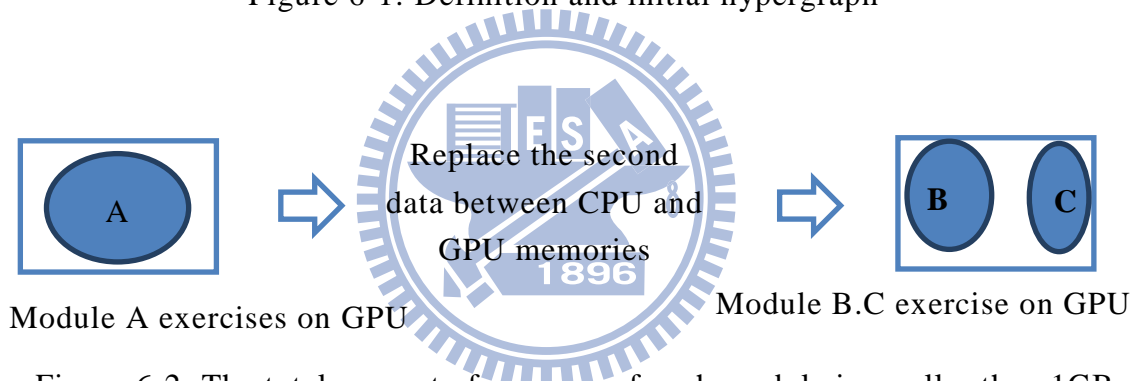


Figure 6-2: The total amount of memory of each module is smaller than 1GB

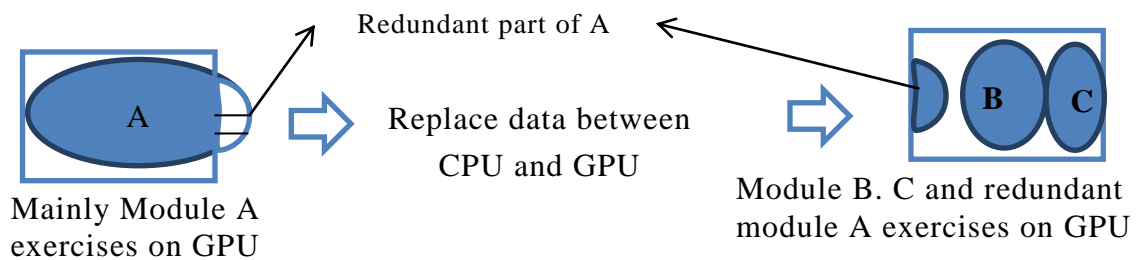


Figure 6-3: Module A is larger than 1GB

What if one of modules is larger than the maximum amount of global memory on a GPU? A casual solution is exchanging information between modules in each time steps. We could see Fig6-3 as an example. First of all, the program partitions the

larger module A into main and redundant parts, minimize the size of outline between these two parts, and copy the duplication between these two parts. Then the program starts to exercise mainly module A in N-body simulation. Since we also copy the duplication, the error between main and redundant parts of module A can be avoided. Next, global memory on a GPU substitutes for next required information, then module B, C and redundant part of module A are fulfilled and operated. Repeat the above steps until simulation terminates. Though the program can frequently substitute information between memories to avoid errors, eternally accessing data cause fiercely latency and lead to worst executed runtime. However, this situation is inevitable under memory limitation. Hence, we also provide second solution which transfers data by blocks instead of time steps and reduce frequently substitutions. The second solution may impact final solutions when main and redundant part of module A still have connections. To avoid likely inaccuracy, the number of connections between these two parts is as few as possible, which could handle by partitioning in the beginning. Consequently, the second solution runs faster than the first solution but gets worst solutions. It depends on the designer's choice.

3.6 Inter-cell Force Modeling

Electronic circuits usually contain both 2-pin and multi-pin nets. Various models have been proposed to replace multi-pin nets by a group of 2-pin nets. For modeling forces in simulating, our research adopts the traditional model which replaces each net by a clique [15]. For example, for the 4-pin net in Fig.7-1, the clique model is shown in Fig.7-2.



Figure 7-1: Multi-pin nets

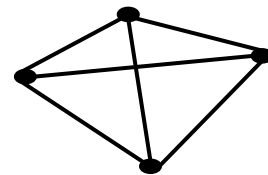


Figure7-2: Clique models for Fig.7-1

N-body simulation focuses on motions between cells instead situation of nets. To model the forces between cells, the given hypergraph is translated into a specific format. Each cell is affected by its own set S_i which is defined as a set of adjacent cells and pins to cell c_i . And n_i is the total number of elements in the set S_i . This following translation model is used for calculating mutual interactions in N-body simulation. For example, Fig.8 illustrates the problem translation into our required format. Fig.8-1 shows a given hypergraph $G=(C, Net)$. Fig.8-2 shows the translation of the hypergraph by using the clique model, and then the interactions between cells are shown. Such as Cell B in Fig.8-3, Cell B is affected by the own set S_B that contains adjacent cells and pins including pin2, cell C, cell D, cell D and cell E.

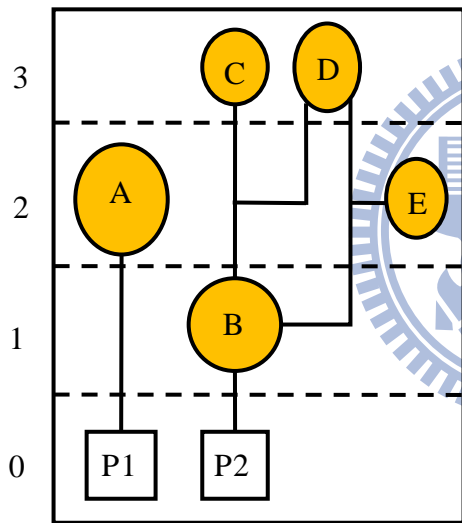


Figure 8-1: A hypergraph $G=(C, Net)$ and its initial solution

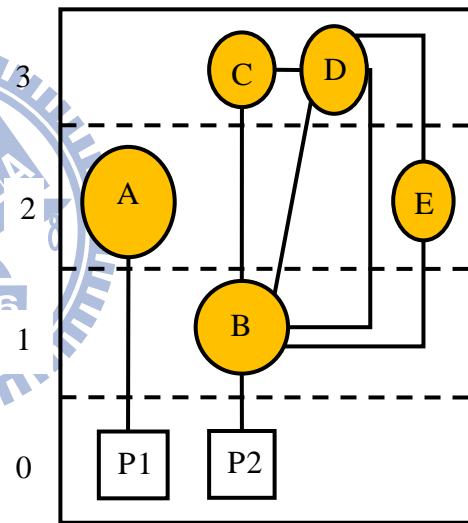


Figure 8-2: The given hypergraph translates into clique model

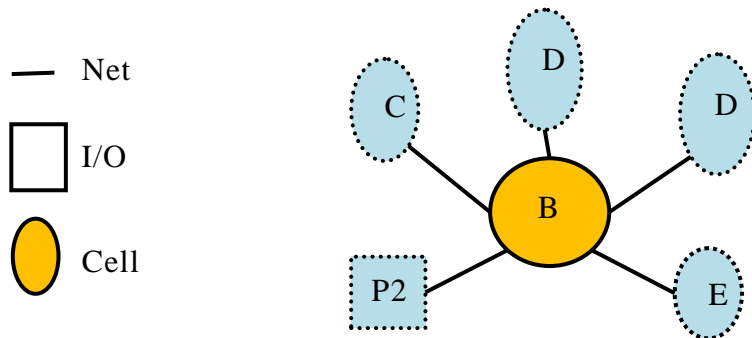


Figure 8-3: Cell B is affected by the set S_B and $n_B = 5$.

Chapter 4

Case Study: 3DICs Partitioning

The partitioning problem formulations and the related works are presented in Chapter 4. The related works are focused on the partitioning algorithms which have been proposed in the past decades. And some symbols used in the following content are also introduced.

4.1 Introduction Of the 3DIC Partitioning

Moore's Law enables the exponential growth of the chip scale. 3DIC technology enables the vertical integration of a large scale system onto the multiple layers of the same die or the same package [16]. The multilayer structure of the 3DIC technology makes it even more computational expensive for EDA tools to achieve optimization goals. The future EDA tools are required to handle the sheer amount of system complexity which requires enormous computation power to sustain the satisfactory performance. Parallel computing has been considered as a solution to meet the soaring requirement [17]. Multi-core architectures, such as general purpose graphic processing unit (GPGPU), have become the mainstream of the future computing systems. Through expanding parallelism in applications, GPGPU can exploit parallelism and achieve superior performance.

Partitioning is in the first step of physical design optimization flow [18]. For a 3DIC, partitioning divides cells into K sub-groups and assigns these groups to different layers. The signals between layers are connected by the Through Silicon

Vias (TSVs) which generally occupy larger chip area than conventional vias. Thus the optimization result of 3DIC partitioning determines the total number of TSVs between layers and has significant impact on several imperative design issues such as number of TSVs, footprint area constraint, timing, thermal distribution, as well as the fabrication yield. To reduce the cost, the goal in this study case is to minimize the number of TSVs under the footprint constraints.

This thesis proposes FDPrior, a highly parallel 3DIC partitioning algorithm based on force-directed scheme with prioritized layering mechanism. FDPrior is among the first to adopt a non-heuristic technique by using force-directed [19] methods to approach 3DIC partitioning problems. These force-directed methods extend the conventional concepts of direct N-body simulation. Chapter 5 will introduce the flow of FDPrior in detail.

4.2 Related Work: Partitioning Algorithms

Partitioning has been extensively researched during the past decades. To resolve partitioning problems, various approaches have been proposed. We have surveyed the following four related works.

The first group of algorithms adopted heuristic approaches to enable easy implementation and return acceptable solution quality. The representative example of this group is the FM algorithm. FM was first proposed by Fiduccia and Mattheyses, and is widely used in solving various partitioning problems. FM used a greedy-like approach to move the cell which could return the maximum gain. However, the runtime of FM gets significantly worse with the increasing number of cells. Besides, the algorithm itself is prone to fall into a local optimum and returned a sub-optimal result.

The second group used deterministic approaches, such as ILP (Integer Linear Programming), to find out the optimal solution. Jiang et.al [20] used the ILP approach to formulate the 3DIC partitioning problem and was able to find the minimum number of TSVs with constrained footprint. The downside of this approach was the extremely long run time which was impractical for a system with a large number of cells.

The third group used multilevel approach to solve the partitioning problem hierarchically. hMetis [21] is one of the most widely used algorithm in this group. hMetis started by repeatedly combining cells into groups of cells (this process called coarsening). When the numbers of cells got smaller and reached an acceptable level, hMetis performed FM partitioning followed by un-coarsening the cell groups to a finer-grain level. The process was repeated until all the created cell groups were un-grouped to the finest level.

With the soaring system complexity in the future 3DIC design, parallel processing on many-core architecture is considered as a solution to enable the scalable computing capability for future EDA tools. PP3D was proposed speed up the 3DIC partitioning by adopting parallel processing scheme. PP3D extended the fundamental ideas of FM algorithm, and added the ability to identify the independent cells which could be moved simultaneously. Compared with the original FM, PP3D achieved one order of runtime speedup while returning the similar solution quality.

4.3 3DIC Partitioning Problem Formulation

This subsequent section will introduce how the 3DIC partitioning problem is modeled for FDPrior algorithm. The definitions of the partitioning problem and variables are described here.

4.3.1 The Structure of 3DICs

Fig.9. shows a simple example of the structure of a 3DIC [22]. All the layers are stacked up vertically. A TSV_IO connects the layer_1 to the bump of a package pin. A TSV, which consists of a TSV_CELL and a TSV_LAND, connects signals between two internal adjacent layers. A TSV_CELL connects to the upper metal layer and a TSV_LAND is a landing pad placed on the lower metal layer. In the partitioning, each TSV is modeled as a cutline between the neighboring layers. The total number of TSVs in this thesis contains both the TSV_IOs and TSVs.

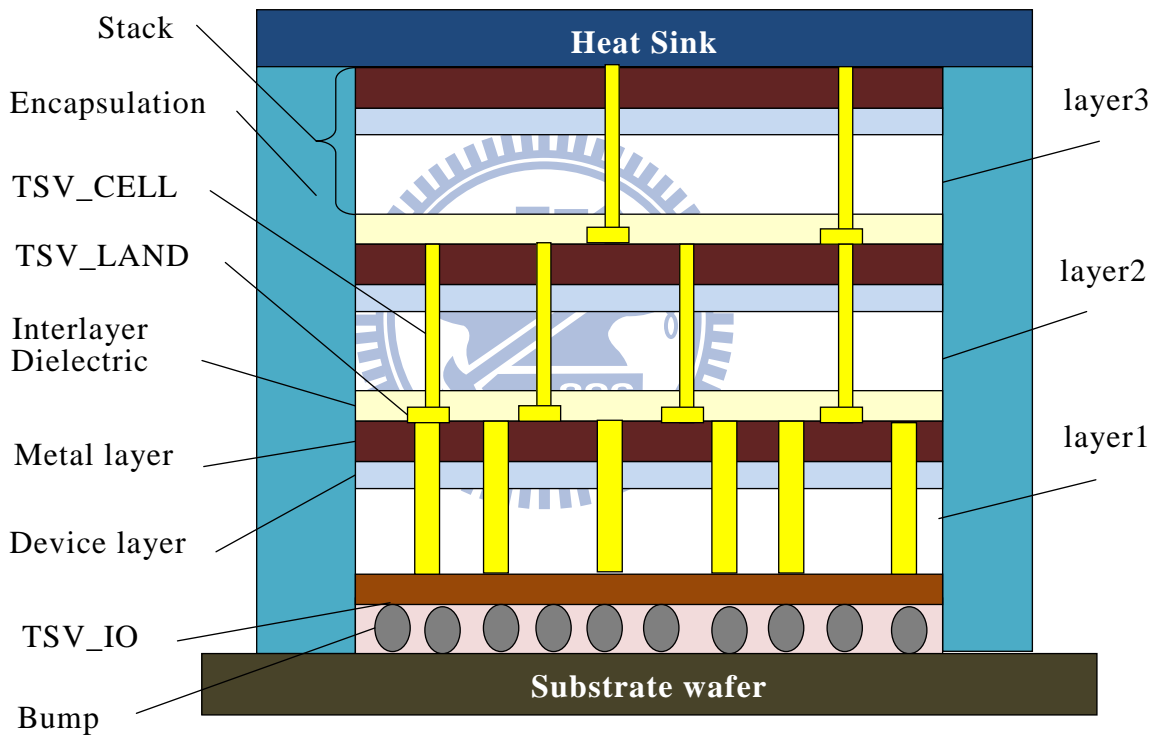


Figure 9: The structure of a 3DIC for vertical interconnects.

4.3.2 Variable Definitions

1. A circuit is modeled by a hypergraph $G=(C, Net)$ where with a set of nets Net that connect to two or more cells and a set of cells C that each cell $c_i \in C$. And A_{cell_i} is the given area of cell c_i .

2. A_{layer_j} is the total area of all cells in layer_j and A_{IO} is the total area of the TSV_IO cells. A_{avg} is the average layer area of a 3DIC. The area constraint of each layer is defined as follows:

$$A_{avg} = (\sum_{j=1}^K A_{layer_j} + A_{IO})/K \quad (3)$$

$$A_{max} = A_{avg} * (1 + F_{ub}/100) \quad (4)$$

$$A_{min} = A_{avg} * (1 - F_{ub}/100) \quad (5)$$

The maximum and minimum bounds of layers are set as A_{max} and A_{min} respectively. F_{ub} is a given constant that decides the range of area bounds. The overflow area is that A_{avg} multiplies the ratio of F_{ub} . And the minimum and maximum area bounds are the average layer area of a 3DIC subtracted or added this overflow area.

3. Equation 6 defines the displacement of the cell c_i . Where x_i represents the current position of c_i , x'_i represents the position of c_i from the previous optimization iteration, and Δx_i is the displacement of c_i between two iterations.

$$x'_i = x_i + \Delta x_i \quad (6)$$

Δx_i , x_i and x'_i are all in the z-direction. The sign of Δx_i guides the direction of cell c_i .

4. During the optimization, each cell can be in either the fixed state or mobile state. A cell in the mobile state can move freely in response to the inter-cell forces while a fixed cell stays at the current position.

4.3.3 Problem Description

The 3DIC partitioning problem can be formulated as a hypergraph multilayer partitioning problem. Given the hypergraph of a netlist and area constraint ($A_{min} \leq A_{layer_j} \leq A_{max}$), a partitioning algorithm divides the set of cells C into K layers ($K \geq 2$) with minimum number of TSVs.

Chapter 5

FDPrior Algorithm

Chapter 5 introduces our proposed algorithm, FDPrior [23]. The main contributions of FDPrior could be categorized into three folds: 1) massive algorithmic parallelism exploited by multi-core GPGPU computing systems 2) a force-directed approach to achieve high result quality 3) bottom-up prioritized layer construction to minimize synchronization overhead.

FDPrior enables a massive computation parallelism by N-Body simulation method which simulates motions of all cells independently. N-Body simulation phase adopts a non-heuristic approach of force-directed method to simulate forces among cells. However, N-body simulation needs fierce mathematical computations for interaction of N particles among a dynamic system. Such N-body problems encounter a formidable challenge to the numerical analysis and computer hardware. Hence, the applications of N-body simulation are particularly suitable for modern multi-core platforms. In partitioning problem, conventional approaches usually define the layer space at first and then optimize solutions on pre-defined layer spaces. However, in parallel algorithm, this method would result in much synchronization which is used to obtain information among cells, such as the distribution. In third fold, FDPrior does not define the layer spaces in the beginning. Instead, the layer space is gradually stacked up by the bottom-up layer construction. This innovative method can significantly reduce the number of synchronizations and gain faster runtime.

In addition to returning better results, FDPrior is designed to take advantage of the scalable computing capability enabled by multi-core computing systems. When

compared with PP3D [24] and the conventional FM [25] algorithms, FDPrior can achieve up to 3.35X and 303.66X times faster runtime on ISPD98 benchmark. Not only finishing the execution faster, FDPrior also returns better results which are in average 7.71X and 5.95X better in terms of total TSV numbers.

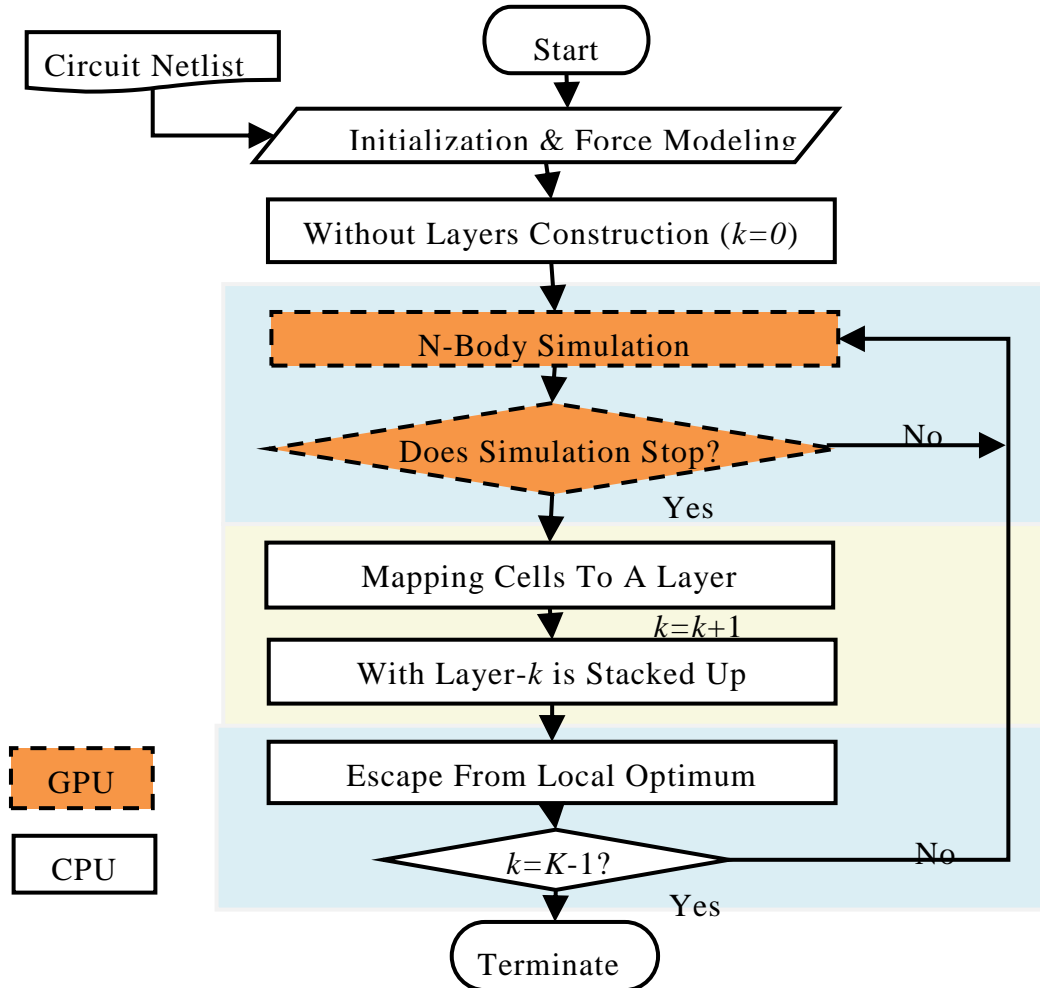


Figure 10: The flow chart of FDPrior

5.1 FDPrior algorithm

The algorithm flow is illustrated in Fig.10. The orange parts implement on the GPU platform, as well as the white parts execute on CPU. The algorithm is composed of three phases, including N-body simulation, mapping cells to a layer, and escape

from local optimum. N-Body simulation phase simulates the movement of each mobile cell based on the forces impacted on it. The second phase picks appropriate cells and maps them to a layer. The first two phases limit the optimization search within a regional area and could fall into a local optimum. Thus the third phase enables a mechanism to escape from local optimum through disturbing the unfixed cells. The above phases will be repeated $K-1$ times to determine the lower $K-1$ layers. The remaining mobile cells after $K-1$ iterations are directly placed into the layer_ K . Details of each phase are discussed in the below sections.

5.1.1 Phase1: N-body Simulation

FDPrior employs the non-heuristic approach to calculate the forces. N-body simulation numerically approximates the motion of each cell in a system. Each cell in a netlist is modeled as a body and be moved independently. Consequently, there has massive parallelism and exploits by the GPGPU platforms. We separate the force into two fundamental components. First, the hold force will give each cell a tendency to stay in the current position. The second component is an attractive force, which is induced by the inter-cell forces to move cells to towards the equilibrium position. These two forces pull each other until the total force is zero; and the system reaches a static equilibrium.

5.1.1.A Hooke's Law

Many problems seem new but actually not. The quadratic placement techniques had been used for the placement problem in a long time. We can refer the previous methods on placement and make modifications for solving partitioning in 3DICs. Accordingly, FDPrior adopts old solutions of the placement problems and assumes the Hooke's law to describe the motions of force. In quadratic placement problem, many

algorithms use the springs or extra forces by Hooke's law to obtain better solutions, e.g. Quinn published in 1975 [26]. This method is usually called force-directed approach in the EDA field.

Simulating mobile cells in a free space may appear vibration when applying gravitational forces that describes in Section 2.2. In our experience, the vibration could take a long time to reach a stable state by using Equation 1 as the force equations. We practice the general rectilinear motion formulae to describe the motion along a straight line. Since there are not frictions in simulation, the accumulated speed keeps each body moving even after they have arrived the meeting point, and results in a two-body vibration situation. According to practical implementations, the vibration mode can rarely provide improvement on the solutions. In the simulation, the vibration mode happens when the direction of a body's acceleration is opposite to its velocity. For this reason, spending too much time on the simulation of the vibration mode is unnecessary and considered redundant.

In physics, Hooke's law of elasticity is an approximation that expresses the extension of a spring with the load applied to it. Hooke's law simply presents that strain is directly proportional to stress. The common application of Hooke's law is spring application. To avoid vibration and obtain quick convergence, FDPrior adopts the forces exerted by the springs, which are defined by Hooke's law. Hooke's law states mathematically in Equation 7 where F is the restoring force exerted by the spring.

$$F = -k\Delta x \quad (7)$$

Where Δx is the displacement of this spring's end from its equilibrium position; and k is the spring constant. A negative sign on the right hand side of Equation 7 shows always opposite direction between displacement and restoring force. FDPrior perform force-directed method to solve equations of forces instead by linear solver.

5.1.1.B Hold Force

The hold force is defined in Equation 8. A_{cell_i} , defined by the area of cell c_i , is the spring constant which affects the strength of the hold force.

$$F_i^{\text{hold}} = -A_{cell_i} \Delta x_i \quad (8)$$

The negative sign presents this cell want to stay in the current position. When the area of a cell is large, this cell has larger momentum and is more difficult to be moved.

5.1.1.C Attractive Force

Fig.11 illustrates the attractive force in z-direction on cell c_i by a spring joining cells c_i and c_j [27]. In a hypergraph, all of the 2-pin nets are exerted by the stretched springs in accordance with clique model.

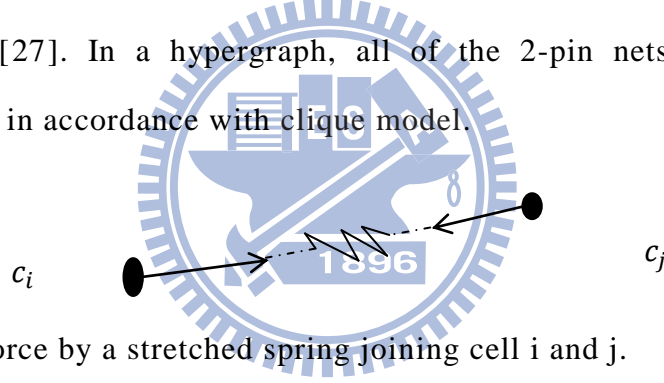


Figure 11: The force by a stretched spring joining cell i and j.

The attraction force between two bodies forms a tendency to pull the bodies closer to each other at every simulation step. The attractive force between cells is defined in Equation 9. The final attractive force is the accumulated forces impacted on a cell c_i .

$$F_i^{\text{attract}} = \sum_{j=1}^{n_i} F_{i,j}^{\text{attract}} = \sum_{j=1}^{n_i} (x_j - x_i), c_j \in S_i \quad (9)$$

In Chapter 3, we mentioned that FDPrior adhere the clique model to modify the given hypergraph, which is translated from the multi-pin nets to a group of 2-pin nets by clique model. All the weight of each 2-pin net is same and set to unit in a system. The formulation of the force in the clique by Hooke's law is equivalent to Equation 8.

5.1.1.D Total Force

```

1  __global__ void  Dkernel_FDPrior_Force(){
2      const int i = blockDim.x * blockIdx.x + threadIdx.x; // create thread id in GPU
3      bool active= true;
4      float distance = 0;
5      if ( i >= cell_size)          active = false;
6      else if ( dcell_anneal[i]<=0 )active = false;    //these fixed cells in the system
7      if ( active ){
8          float nowlayer = dcell_layer[i];    //the current position of cell with index
9          for (int j=dc2c_nbegin[i];j<dc2c_nbegin[i+1];j++)    //the identified set i
10             distance+= dcell_layer[ dcell2cell[j] ] - nowlayer;
11             dcell_force[i]= distance;    // the final attractive force of cell with index i
12     }
13     __syncthreads();
14     if ( active ){
15         distance = __fdividef(dcell_force[i],dcell_area[i]);//calculate displacement
16         dcell_layer[i]+= distance;    //update the new position of cell i
17         if( dcell_layer[i] < 0 ) dcell_layer[i] = 0;
18     }
19 }

```

Table 3: The code of the calculated forces in the N-body simulation phase

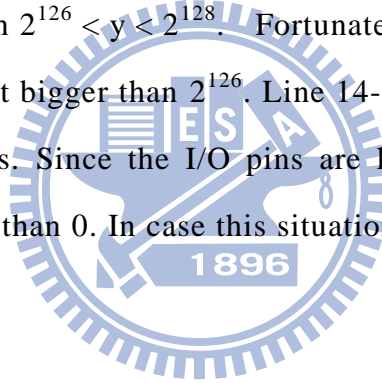
The total force is the sum of the hold force and attractive force. The new cell positions are efficiently computed by solving Equation 10 for Δx_i .

$$F_i^{total} = F_i^{attract} + F_i^{hold} = 0 \quad (10)$$

The N-Body Simulation phase terminates when the number of iterations reaches the threshold level or the total number of iterations exceeds a configured limit. In each layer space, configured limit is 2000 in our experimentation. The threshold level is defined as r_{dis}^* distribution ratio where r_{dis} is an empirical number. The range used in

this algorithm is $r_{dis} = 0.95$. And the distribution ratio is the total number of mobile cells which fall in the constructed layers divided by the rest of the mobile cells. In other words, r_{dis} * distribution ratio means that the majority of cells exercise enough and already drop to the lower layer by the connected nets.

Table 3 shows the calculations of forces in the N-body simulation phase. Line 6 percolates already fixed cells in a system. Line 7-12 calculates each attractive force with cell c_i by creating a thread with index i . Because throughput of single-precision floating-point division takes some cycle, we use `__fdivdef` function to accelerate for division, shown in line 15. This `__fdivdef(x,y)` function means x divided by y and provides a faster math version. Both the regular floating-point division and `__fdivdef(x,y)` have the same accuracy. But `__fdivdef(x,y)` delivers a result of zero and cause inaccuracy when $2^{126} < y < 2^{128}$. Fortunately, the areas of every cells in the ISPD98 benchmark are not bigger than 2^{126} . Line 14-18 arranges the data and updates the new positions of cells. Since the I/O pins are located in the bottom layer, the positions cannot be lower than 0. In case this situation occurs, we add a refinement in line 17.



5.1.2 Phase 2: Mapping Cells To A Layer

In the Mapping Cells To A Layer phase, the layer space is constructed by gradually stacking up cells from the bottom of a layer. Since it is a bottom-up approach, cells at lower positions have higher priorities to be included in this layer. When more cells are included in this layer, the area of this layer will increase. And `FDPrior` will search the best layer boundary (with minimal outline) between the minimum layer area bound (A_{min}) and maximum layer area bound (A_{max}) which were defined in Chapter 3. These cells which are mapped into this layer will be changed into the fixed state, while others remain in mobile states.

In other words, FDPrior will gradually search cells which are located in lower positions and straightly put them in this layer until the area of this layer reaches the minimum layer area bound. For example, these cells with index 1 ... p are successively mapped into a layer. Then FDPrior estimates the cutline and save data in the buffer when each cell moves until the area of this layer reaches the maximum layer area bound. Suppose these cells with index p+1 ... q are already handled and each cell has its own cutline. Finally, the program will seek out the minimum cutline between cells with index from p+1 to q, as well as set the appropriate cells into fixed states. If the cell with index g has the minimal cutline, cells with index g+1 ... q will set to freedom in a system. In conclusion, only these cells with index 1 ... p, p+1 ... g are firmly mapped into these layer and turned states into the fixed states.

In the program beginning, there does not define any layer spaces. And all the cells of the given hypergraph are in the mobile states. Each layer space will be stacked up after repeating executes the Mapping Cells To A Layer phase. Until the lower $K-1$ layers all are successively implemented, the remaining mobile cells in the system will be directly placed into the layer K .

5.1.3 Phase 3: Escape From Local Optimum


The previous two phases limit the optimization search within a regional area and the solution may fall into a local optimum. FDPrior adds an approach to escape from the possible local optimum by disturbing certain mobile cells based on Equation 10.

$$\Delta x_i = (F_i^{attract} / A_{cell_i}) + (n_i / n_{avg}) \quad (11)$$

The parameter n_{avg} is the average of the summation of n_i .

5.2 Experimental Results

The target GPGPU platform of FDPrior is Nvidia Geforce 9800GT [28]. All the experiments are conducted on a 2.66GHz Intel® Core(TM) i7-920 CPU sprinting CentOS 5.5 with 6GB of memory. This research adopts the ISPD98 benchmark suite to evaluate the algorithms [29]. ISPD98 contains a wide range of varieties, which is listed in Table 4. The F_{ub} indicates the unbalance of layers and the area footprint constraint. The number of desired layers and F_{ub} are different when considering characteristics in every case are distinct. If the chip area is immense, we donate this chip more freedom in the area bounds, such as the chip area in ibm18 is larger then the given constant F_{ub} will be slightly great. The numbers of threads per block are 256 in all cases when considering the limitation of registers. Consequently, the total number of threads that we implemented in the GPGPU platform is 256* the number of blocks per grid.



Bench	#Block	#cells	#nets	#I/Os	#layers	F_{ub}
ibm01	56	12506	14111	246	4	10
ibm02	78	19342	19584	259	4	10
ibm03	108	22853	27401	283	4	10
ibm04	126	27220	31970	287	4	10
ibm05	112	28146	28446	1201	4	10
ibm06	137	32332	34826	166	4	10
ibm07	189	45639	48117	287	5	12
ibm08	200	51023	50513	286	5	12
ibm09	239	53110	60902	285	5	12
ibm10	295	68685	75196	744	5	12
ibm11	319	70152	81454	406	5	12
ibm12	303	70439	77240	637	5	12
ibm13	390	83709	99666	490	6	15
ibm14	598	147088	152772	517	6	15
ibm15	730	161187	186608	383	6	15
ibm16	743	182980	190048	504	6	15
ibm17	742	184752	189581	743	6	15
ibm18	823	210341	201920	272	6	15

Table 4: Characteristics in ISPD98 benchmark

Fig.12 compares the average number of TSVs generated by FDPrior, PP3D [22] and MLFM algorithms. PP3D leverages the fundamental ideas of FM algorithm, and adds the ability to identify the independent cells. MLFM represents the extended conventional FM algorithm which considers multilayer partitioning. FDPrior obtains much better qualities than PP3D and MLFM in every benchmark. The experimental numbers are the average of 30 runs. The average numbers of TSVs which execute on the algorithms in ISPD98 benchmark are listed in Table 5.

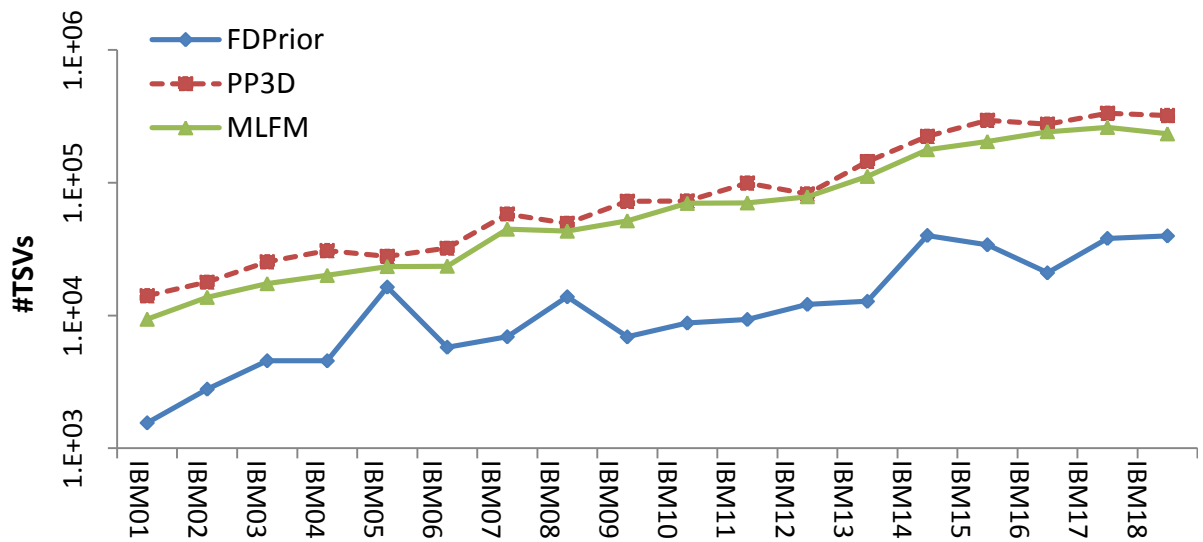


Figure 12: Comparison of average solutions on the algorithms.

When comparing with PP3D and MLFM, FDPrior demonstrates an average of 7.71X and 5.95X better solution quality respectively. Since PP3D introduces the primary ideas of FM algorithm, the solutions of PP3D is a bit of near to the solutions of MLFM. FDPrior accomplishes an absolutely different algorithm by using N-body algorithm and creates coarser grain than others. That is the reason that FDPrior can perform better solution qualities.

From the distribution of 30 runs, FDPrior is more stable on solution qualities than PP3D and MLFM algorithms. Fig.13 shows the solution qualities of PP3D are distributed across a wider range than FDPrior in ibm04. FDPrior also achieves 3.35X

and 303.66X runtime speedup than PP3D and MLFM, as illustrated in Table 6. Because PP3D also uses parallel techniques to achieve improved and scalable runtime, the runtime enhancement that compared with PP3D is not as significant as MLFM algorithm. Since FDPrior is the coarsest between these algorithms, the enhancements of runtime can reach more than the number of multiprocessors in a GPU.

Bench	FDPrior	PP3D	Improve	MLFM	Improve
ibm01	1554	14061	9.05 X	9379	6.04 X
ibm02	2788	17839	6.40 X	13711	4.92 X
ibm03	4562	25411	5.57 X	17380	3.81 X
ibm04	4558	30815	6.76 X	20051	4.40 X
ibm05	16377	27886	1.70 X	23395	1.43 X
ibm06	5769	32156	5.57 X	23532	4.08 X
ibm07	6917	58173	8.41 X	44715	6.46 X
ibm08	13841	49523	3.58 X	43202	3.12 X
ibm09	6902	72548	10.51 X	51703	7.49 X
ibm10	8772	73016	8.32 X	69933	7.97 X
ibm11	9336	99497	10.66 X	70509	7.55 X
ibm12	12125	82462	6.80 X	78490	6.47 X
ibm13	12780	144566	11.31 X	111705	8.74 X
ibm14	40104	223969	5.58 X	177402	4.42 X
ibm15	34110	294835	8.64 X	204159	5.99 X
ibm16	20969	275804	13.15 X	242433	11.56 X
ibm17	38093	333851	8.76 X	261111	6.85 X
ibm18	39787	320939	8.07 X	233465	5.87 X
Average TSVs comparisons			7.71 X		5.95 X

Table 5: The average number of TSVs execute on the algorithms in each case.

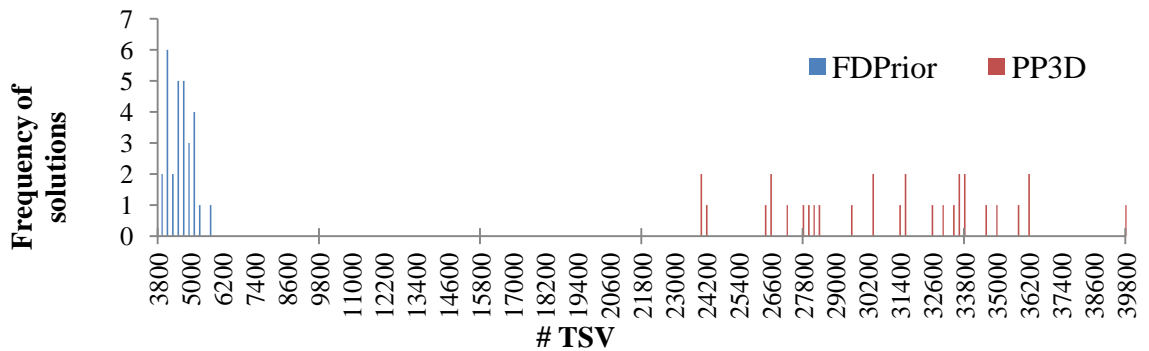


Figure 13: The distribution of solution qualities of algorithms in ibm04

Bench	FDPrior(s)	PP3D(s)	speedup	MLFM(s)	speedup
ibm01	3.333	7.086	2.13 X	173.639	52.10 X
ibm02	8.955	21.825	2.44 X	424.350	47.39 X
ibm03	7.541	21.896	2.90 X	587.212	77.87 X
ibm04	10.941	47.335	4.33 X	787.745	72.00 X
ibm05	8.103	49.416	6.10 X	1122.722	138.56 X
ibm06	9.374	40.063	4.27 X	1321.312	140.95 X
ibm07	17.813	75.340	4.23 X	3203.600	179.85 X
ibm08	39.532	185.079	4.68 X	4112.500	104.03 X
ibm09	25.579	161.444	6.31 X	5139.150	200.91 X
ibm10	42.190	240.582	5.70 X	12679.400	300.53 X
ibm11	37.980	182.962	4.82 X	9515.250	250.53 X
ibm12	52.766	262.594	4.98 X	10931.650	207.17 X
ibm13	68.717	98.832	1.44 X	29764.550	433.15 X
ibm14	168.460	231.254	1.37 X	87466.400	519.21 X
ibm15	231.977	303.262	1.31 X	133523.250	575.59 X
ibm16	271.378	311.795	1.15 X	178302.167	657.03 X
ibm17	280.808	272.444	0.97 X	224073.400	797.96 X
ibm18	286.660	326.624	1.14 X	203829.800	711.05 X
Average runtime speedup			3.35X		303.66 X

Table 6: Comparison of runtime on the algorithms

(Speedup = execution time of PP3D or MLFM/execution time of FDPrior)

The benchmark in ibm05 is a special case that has many I/O pins which may push connected cells to the bottom layer and return undesired solutions. Even though this case limits the improvement, FDPrior still obtains better quality on TSVs and runtime. We extend FDPrior to manage these types of special cases by referring well-known multilevel mechanism. The solution for this special case is mentioned in Chapter 5.

5.3 The Discussion of FDPrior

This section discusses three issues on FDPrior algorithm. The first discussion focuses on the distinguishing characteristic of algorithms, which caused dissimilar experimental results. Secondly, the distributions of all layer spaces in a 3DIC are addressed. Finally, we investigate the difference of parallel and sequential schemes in N-body simulation phase, as well as provide experimentations.

5.3.1 Comparisons Between algorithms

PP3D searched the independent groups of cells from the given circuit and parallelized these groups on the GPU platform. On account of adopting the essential FM algorithm to execute motions of these groups, solutions earned from PP3D are quite similar to solutions obtained from FM. Besides, PP3D accomplished the parallel techniques and performed coarser grain than FM. Hence, PP3D achieved acceleration than FM algorithm.

Compared with PP3D and MLFM, FDPrior was a completely different algorithm including data structure, frame work of algorithm and parallel methodology. The attractive forces in our algorithm are able to pull related particles together. Cells in a circuit are separated into groups by attractive forces to achieve the purpose of minimizing TSVs. From Section 4.4, FDPrior demonstrated better performance not only on solutions but also on runtime. Besides, FDPrior possessed massive parallelism than PP3D and MLFM by using N-body simulation. Unlike only the independent groups are paralleled on PP3D, entire cells in a system are calculated and paralleled on FDPrior. Though PP3D also performed the parallel technique, FDPrior still was able to speedup runtime than PP3D. And that is also the reason that the runtime improvement of FDPrior could be achieved 303.66X than FM with only 64 multi-cores.

5.3.2 The Distribution Of Layer Spaces

In virtue of using bottom-up method, the layer spaces in each layer of a 3DIC are somewhat of unbalance. This issue is inevitable when FDPrior fulfills the bottom-up prioritized layer construction in the mapping cells in a layer phase. Table 7 provides the area distribution of all layers in ISPD98 benchmark. μ is the mean(equal as A_{avg}).

σ is the standard deviation of all layer areas in the circuit. And CV presents the area coefficient of variation and gauges the standard deviation ration to mean.

Bench	ibm01	ibm02	ibm03	ibm04	ibm05	ibm06
Layer1	1018026	2165707	2409785	2303008	1050958	2055563
Layer2	1140239	2181378	2395257	2432731	1200274	2183326
Layer3	1085824	2119028	2602078	2413369	1226727	2158318
Layer4	985927	1992223	2435761	2145837	993561	2180584
μ (Mean)	1057504	2114584	2460720	2323736	1117880	2144448
σ (standard deviation)	59852	74275	82893	114008	98197	52226
$CV=\sigma/\mu$	5.66%	3.51%	3.37%	4.91%	8.78%	2.44%
Bench	ibm07	ibm08	ibm09	ibm10	ibm11	ibm12
Layer1	2284988	2583134	3378905	9267740	4262161	7440142
Layer2	2301881	2537983	3595548	9372725	4092020	7331389
Layer3	2406695	2958431	3596627	10407371	4407963	7837406
Layer4	2573079	2941901	3627754	9532816	4567842	7388670
Layer5	2263213	2428439	3330479	8953684	3907422	6977242
μ (Mean)	2365971	2689978	3505862	9506867	4247482	7394970
σ (standard deviation)	114697	218382	124913	488419	231693	274361
$CV=\sigma/\mu$	4.85%	8.12%	3.56%	5.14%	5.45%	3.71%
Bench	ibm13	ibm14	ibm15	ibm16	ibm17	ibm18
Layer1	3895623	4665888	5456475	8316207	7888877	5235930
Layer2	4201036	4357280	6025610	8827031	6974373	5634866
Layer3	4243918	4361440	6071698	8950749	7745547	5605915
Layer4	4492202	5211904	6593459	9718766	7496915	5702046
Layer5	4538258	5027520	6779301	9587458	7762127	6084722
Layer6	3690531	5103264	5608401	7192493	4319873	5423082
μ (Mean)	4176928	4787883	6089157	8765451	7031285	5614427
σ (standard deviation)	302969	346131	477210	846429	1248437	261157
$CV=\sigma/\mu$	7.25%	7.23%	7.84%	9.66%	17.76%	4.65%

Table 7: The distribution of all layer areas of FDPrior in ISPD98 benchmark

From Table 7, the average coefficient of variation is 6.33%, which depicts the area distribution of all layer space is not extremely uniform. The coefficients of variation also are related to the given parameter F_{ub} . If F_{ub} is bigger, the coefficient

may be bigger for allowing more unrestrained areas. When F_{ub} is 10 and desired layer is 4, the average coefficient of variation is 4.78% in ibm01 ~ ibm06. The average coefficient of variation is 5.14% in ibm07 ~ ibm12 while F_{ub} is 12 and desired layer is 5. When F_{ub} is 15 and desired layer is 6, the average coefficient of variation is 9.07% in ibm13 ~ ibm18. Even though the unbalance circumstances are unavoidable, we ensure each layer can meet the area bounds ($A_{min} \leq A_{layer_j} \leq A_{max}$).

5.3.3 Parallel And Sequential Schemes on N-body Simulation

When N-body simulation provides coarse grain than other algorithms, the interactive forces are infected by the adjacent cells; nevertheless, there are different appearances between parallel and sequential schemes which may cause dissimilar solutions. Hence, we try to discover appearances between these schemes.

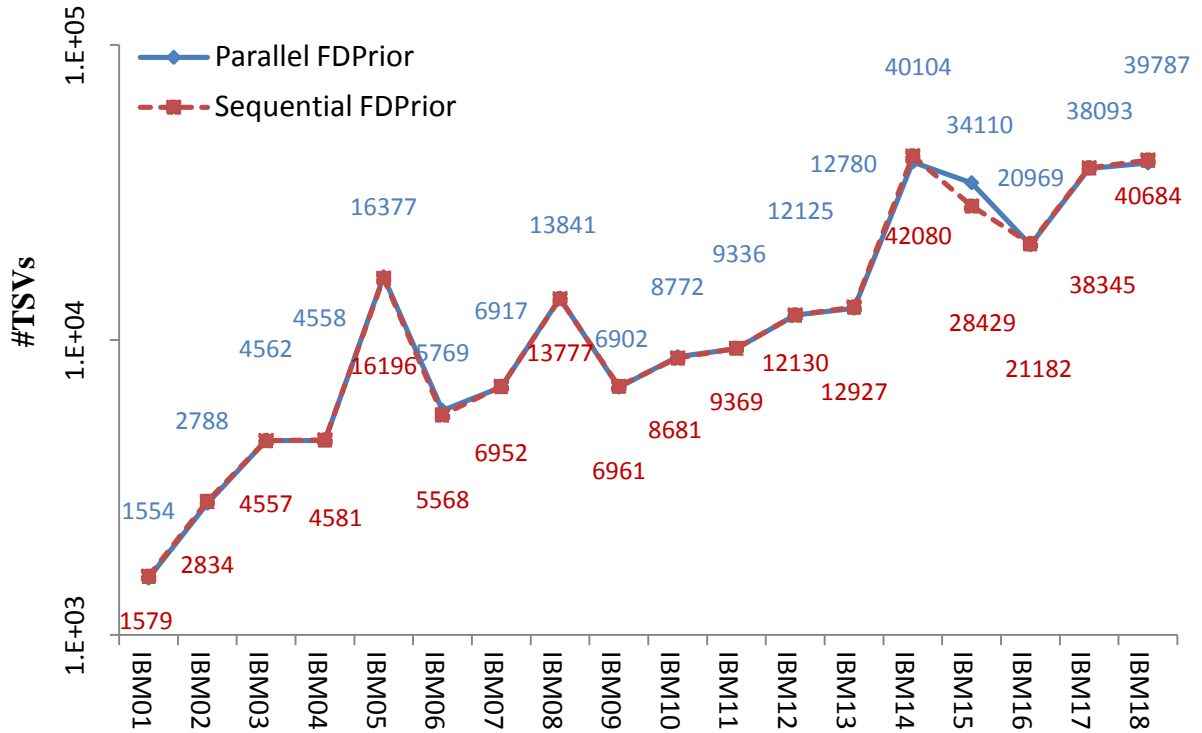
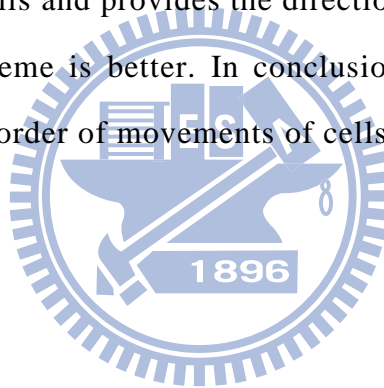


Figure 14: Comparisons of the average solutions between parallel and sequential schemes of N-body simulation

In the sequential scheme of FDPrior, the own forces of a cell must be calculated under the current conditions. For convenience, we test the sequential scheme by moving mobile cells from index 1 to index n sequentially. In addition, FDPrior originally uses parallel concepts that all mobile cells in a system are moved in the meanwhile. That is the operation in the parallel scheme of FDPrior is as same as before.

Fig. 14 illustrates the experimental results in ISPD98 between the parallel and sequential methods in FDPrior. The solutions between the parallel and sequential schemes of N-body simulation are not many differences from Fig.14. There is no doubt that the parallel method is a primary phenomenon of the nature. A scheme just determines the order of cells and provides the direction of motion in the future. Hence, we cannot say which scheme is better. In conclusion, no matter you choose which schemes to determine the order of movements of cells, the solutions are almost equal.



Chapter 6

MFDPrior: Multilevel Of FDPrior

Chapter 6 addresses the structure of multilevel and presents the modified algorithm MFDPrior. MFDPrior uses the basic concepts of multilevel to minimize the number of TSVs, as well as inherits the three main contributions of FDPrior. MFDPrior is designed to take advantage of the multilevel structure. Compared with prior algorithm FDPrior, MFDPrior can achieve up to 1.44X faster runtime and also returns average 1.46X better qualities in the ISPD98 benchmark. In the end, we also discuss the probably exerting influences on MFDPrior algorithm.

6.1 Introduction of Multilevel

There are various proposed researches targeted on the partitioning problem during the past decades. A sequence of coarser hypergraph is formed in these published algorithms. Fig. 15 shows the simple flow of the multilevel mechanism. Coarsening is the process that repeatedly combines vertices into groups of vertices hierarchically, and then the multilevel system of a given hypergraph is constructed. When the size of vertices got smaller and reached an acceptable level, the program begins executing partitioning algorithm followed by un-coarsening these groups to a finer-grain level. The ordinary partitioning algorithm is used to iteratively improve solutions at each level. These processes are repeated until all the created groups are un-grouped to the finest level in a circuit system.

In recent years, 3DIC physical designs are more and more noticeable in the EDA

filed. In the experimental results, the multilevel hypergraph partitioning algorithms could earn better performances than non-multilevel methods [30]. Therefore, we modify our proposed algorithm by adding a multilevel mechanism. In order to discriminate prior version, the proposed algorithm which fulfills the multilevel framework is called MFDPrior. And MFDPrior assumes previously FDPrior as the basic partitioning part.

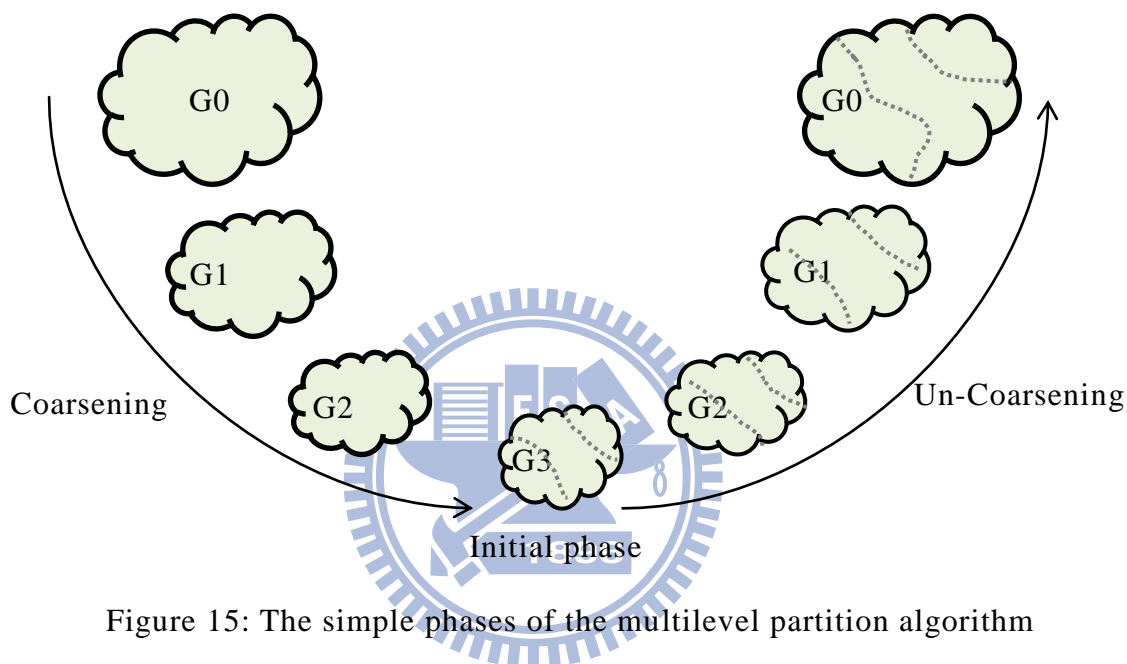


Figure 15: The simple phases of the multilevel partition algorithm

6.2 MFDPrior: The Multilevel Methodology of FDPrior

Fig.16 shows the tendered algorithm flow of MFDPrior. MFDPrior algorithm is composed of three phases, including multilevel coarsening, partitioning and multilevel un-coarsening phases. First of all, multilevel coarsening phase creates the groups of vertices and decides coarser hypergraphs in each level. Hypergraph coarsening assists in reducing the sizes of hyperedges. Besides, multilevel coarsening phase also performs initialization of a solution and construction of force models in the finest level. Secondly, the partitioning phase operates partitions by using modified FDPrior-based or prior FDPrior algorithms. Modified FDPrior-based is no need to

execute precisely and used for reducing runtime. After several levels of coarsening, the program will stop the multilevel coarsening phase and start to enforce the un-coarsening phase. The un-coarsening phase simply unloosens the groups of hyperedges in each level which permit the hypergraph become bigger until as same as the incipient hypergraph. Details of each phase of multilevel mechanism are discussed in the below sections.

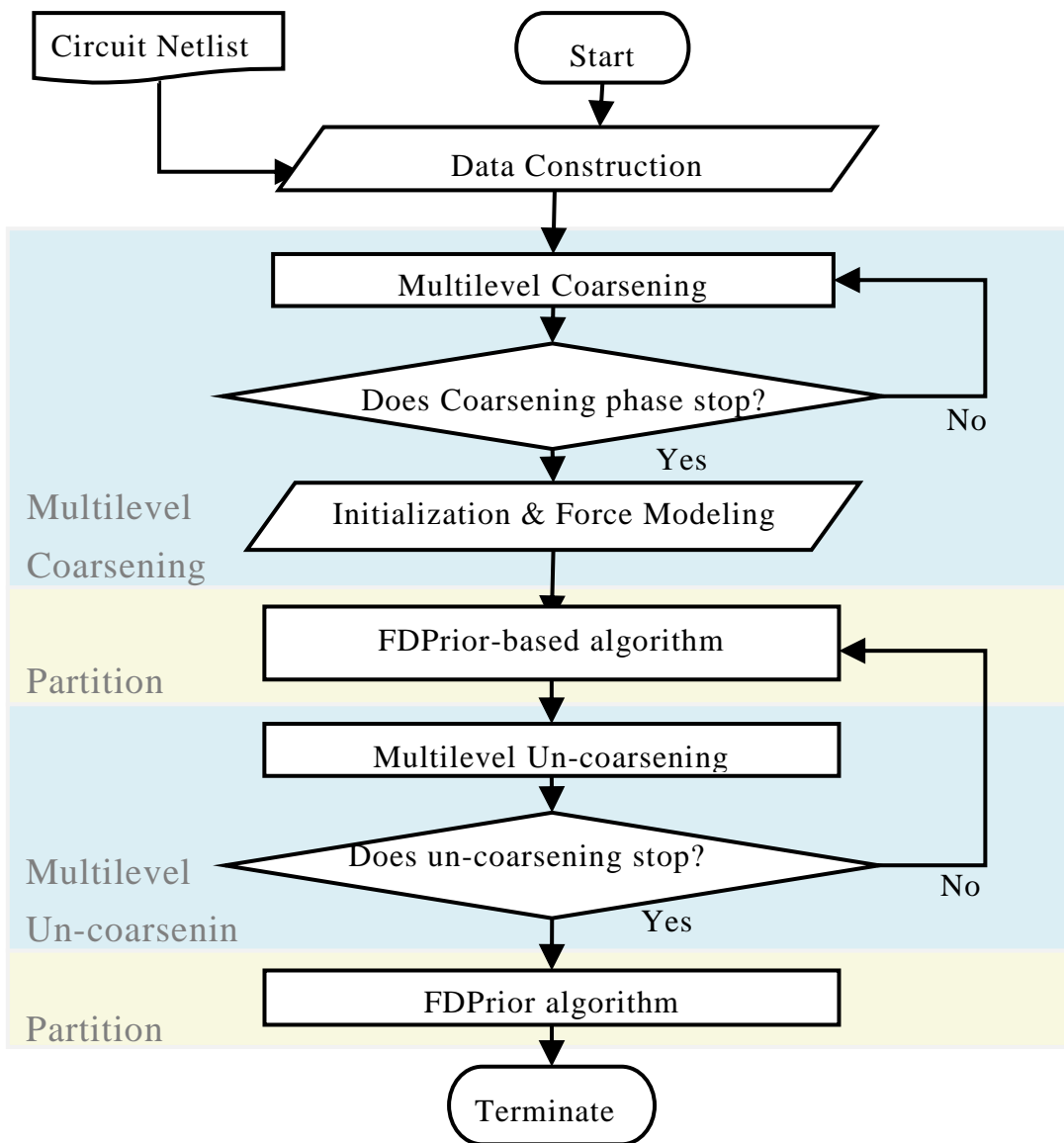


Figure 16: The flow chart of MFDPrior algorithm

6.2.1 Multilevel Coarsening Phase

Multilevel coarsening phase refers to the famous modified hyperedge coarsening (MHEC) scheme [31]. Fig. 17 presents the simple relationship of original hypergraph, as well as coarsens vertices into groups and composes a coarser hypergraph as a netlist of the next level. A black ball is a vertice and a ring is a net. Fig. 17 illustrates the evolution of a hypergraph by using MHEC scheme. MHEC scheme primarily selects which vertices should be merged together and forms a single vertice in the next level. MHEC could successively decrease the size of hyperedges, as well as avoid severely unbalance of the weight of the vertices in coarse graphs.

We defines the maximum area of a group is $0.2 * A_{avg}$ [32]. When the area of a group reaches maximum bound, this group will stop coarsening in this level and stay the present status in the follow-up level. If a cell is larger than $0.2 * A_{avg}$ in the beginning, this cell will absolutely not be included in any groups and maintain as a single vertice all the time. The multilevel coarsening phase terminates until none of the groups could be coarsened or a threshold level is met. The threshold level is specified as $0.084 * \text{the size of } C$. When the hypergraph reaches the coarsest level, the program will begin to initialize an initial solution and assign these groups of the coarsest hypergraph to K layers randomly. These groups which associated to the pins are directly placed to the lowest layer.

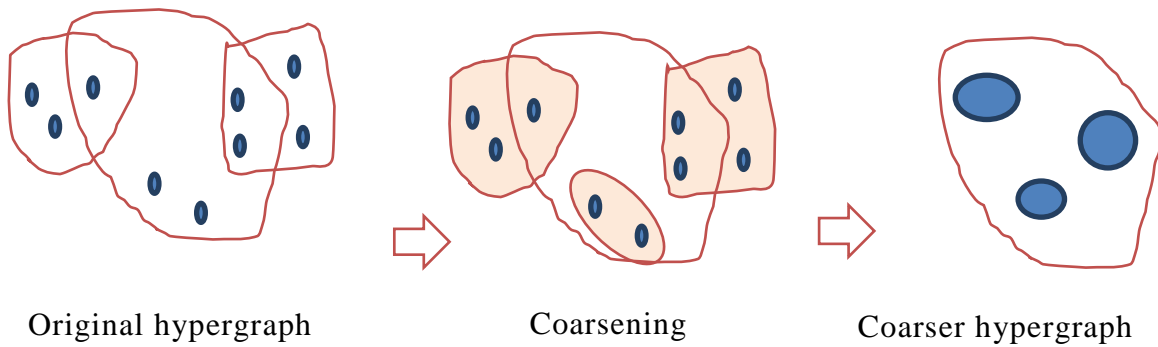


Figure 17: Simple diagram of modified hyperedge coarsening

6.2.2 Partitioning Phase

During the partitioning phase, we separate the partitioning algorithms into two parts: FDPrior-based and FDPrior. FDPrior purely exercises when the hypergraph is un-coarsened into the finest level which is equal to the incipient hypergraph; otherwise FDPrior-based algorithm enforces the other levels. The application of FDPrior is same as mentioned algorithm in Chapter 4.

Transferring memory between GPU and CPU sides is very time-consuming. Because FDPrior runs a lot of iterations in the N-body simulation phase, the transferring time between GPU's and CPU's memory can be effectively hidden. However, running much iteration in simulating is unessential during the coarser hypergraph. Therefore a new approach, called FDPrior-based algorithm which is the fundamental scheme of FDPrior, is addressed.

FDPrior-based algorithm eases the mapping cells into a layer phase instead of the mapping all cells into a 3DIC once. That is FDPrior-based merely performs two phases, including N-body simulation and mapping cells into a chip phases. These two phases execute once individually and do not repeat again. More precisely, mapping cells into a chip is equal to the mapping cells into a layer phase of FDPrior with repetitions, but does not go through more simulation phase. FDPrior-based algorithm can return nice qualities and decrease the execution time during the coarser level.

6.2.3 Multilevel Un-coarsening Phase

An accomplished partitioning of the hypergraph is projected to the next hypergraph. Since the next level has finer hypergraph, more sizes of vertices and provides more freedom in a meanwhile, the partitioning phase can improve the solution qualities. The multilevel un-coarsening phase terminates when the level of primary hypergraph is met.

6.3 Experimental Results

The experimental environment is as same as before mentioned in Section 4.4. The target GPGPU platform of MFDPrior is also Nvidia Geforce 9800GT. And ISPD98 benchmark suit, which contains a wide range of varieties and listed in Table 4, is used to evaluate the algorithms in this thesis.

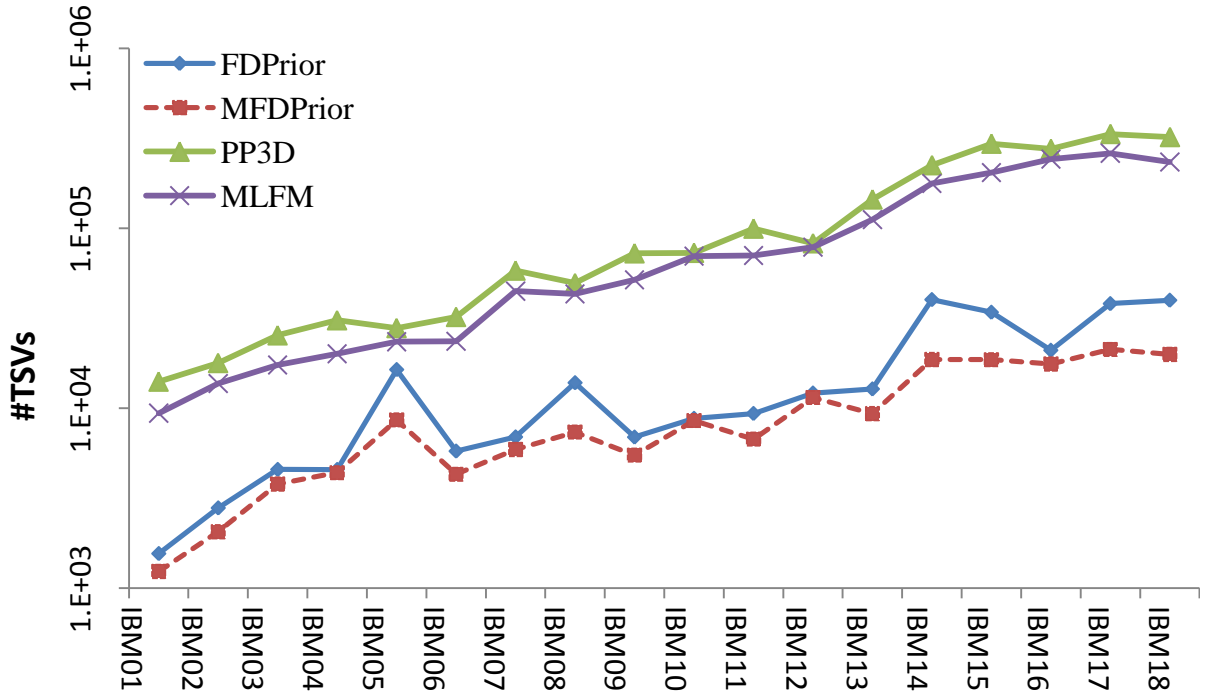


Figure 18: Comparisons of average number of TSVs with FDPrior and MFDPrior

Fig.18 compares the average number of TSVs generated by MFDPrior and FDPrior. The experimental numbers are the average of 30 runs. The experiential results of FDPrior, PP3D and MLFM are same in the lecture 4.4. MFDPrior leverages the fundamental ideas of proposed FDPrior algorithm, and performs the multilevel methodology. MFDPrior obtains much better qualities than FDPrior in almost all cases. Table 8 lists the average numbers of TSVs of MFDPrior and FDPrior in ISPD98 benchmark. When comparing with incipient FDPrior, MFDPrior demonstrates an average of 1.46X better solution quality and earns 1.44X speedup. Since FDPrior had used the parallelism method, the runtime enhancement is not remarkable.

Due to the size of multilevel in ibm02 is a little more, it need more execution time to handle these multilevel structures. Though the average runtime of MFDPrior is slightly bigger than FDPrior in ibm02, the execution time is still faster than PP3D and MLFM. Furthermore, the benchmark in ibm05 is a special case which has many I/O pins. The solutions of FDPrior in ibm05 are not distinguished for qualities, because ibm05 may push connected cells to the bottom layer quickly. Even though this special case limits the enhancement of FDPrior, MFDPrior can obtain better performances under quickly runtime. Generally, MFDPrior can require superior solution qualities under rapidly execution times than FDPrior.

Bench	MFDPrior TSVs	FDPrior TSVs	Improve on TSVs	MFDPrior runtime(s)	FDPrior runtime(s)	Improve on runtime
ibm01	1238	1554	1.26 X	1.916	3.333	1.74 X
ibm02	2057	2788	1.36 X	9.470	8.955	0.95 X
ibm03	3788	4562	1.20 X	4.699	7.541	1.60 X
ibm04	4379	4558	1.04 X	5.440	10.941	2.01 X
ibm05	8605	16377	1.90 X	5.983	8.103	1.35 X
ibm06	4290	5769	1.34 X	6.521	9.374	1.44 X
ibm07	5896	6917	1.17 X	14.089	17.813	1.26 X
ibm08	7366	13841	1.88 X	24.873	39.532	1.59 X
ibm09	5484	6902	1.26 X	17.871	25.579	1.43 X
ibm10	8537	8772	1.03 X	31.014	42.190	1.36 X
ibm11	6731	9336	1.39 X	24.145	37.980	1.57 X
ibm12	11462	12125	1.06 X	34.822	52.766	1.52 X
ibm13	9331	12780	1.37 X	46.917	68.717	1.46 X
ibm14	18650	40104	2.15 X	119.887	168.460	1.41 X
ibm15	18609	34110	1.83 X	167.998	231.977	1.38 X
ibm16	17610	20969	1.19 X	194.814	271.378	1.39 X
ibm17	21215	38093	1.80 X	219.653	280.808	1.28 X
ibm18	19876	39787	2.00 X	229.676	286.660	1.25 X
Average TSVs comparisons			1.46X	Average runtime comparisons		1.44X

Table 8: Comparisons of average solutions and runtime with FDPrior and MFDPrior

6.4 Discussion of Issues

MFDPrior not only inherits the properties of FDPrior but also creates probably occurrences. In these lectures, we share our practical experience and provide discussions about these issues. First one is the distribution topic of MFDPrior that inherits the characteristics from prior FDPrior. Finally, we compare the experimental results with the renowned algorithm.

Benchmark	ibm01	ibm02	ibm03	ibm04	ibm05	ibm06
Layer1	1160642	2134166	2366627	2386170	1055382	2055417
Layer2	1075524	2156400	2331876	2417590	1228538	2269601
Layer3	1124381	2004218	2629262	2438889	1069225	2001606
Layer4	869469	2163552	2515115	2052295	1118375	2251167
μ (Mean)	1057504	2114584	2460720	2323736	1117880	2144448
σ (standard deviation)	112685	64635	119184	157834	68042	117667
$CV=\sigma/\mu$	10.66%	3.06%	4.84%	6.79%	6.09%	5.49%
Benchmark	ibm07	ibm08	ibm09	ibm10	ibm11	ibm12
Layer1	2348790	2619671	3620334	9495844	4566129	7139564
Layer2	2397711	2633680	3540201	9817553	4343167	7304691
Layer3	2313189	2952016	3718369	9959926	4240479	7825616
Layer4	2586682	2867053	3441861	9875606	4282647	7520189
Layer5	2183484	2377468	3208547	8385406	3804986	7184788
μ (Mean)	2365971	2689978	3505862	9506867	4247482	7394970
σ (standard deviation)	131227	202877	174318	582332	248105	252466
$CV=\sigma/\mu$	5.55%	7.54%	4.97%	6.13%	5.84%	3.41%
Benchmark	ibm13	ibm14	ibm15	ibm16	ibm17	ibm18
Layer1	3859282	4356992	5865085	9036549	8072990	5973680
Layer2	4253040	4354304	5877701	9491959	7078425	5685904
Layer3	4228580	4357760	5973489	9033808	7465811	5850787
Layer4	4236985	5505600	6381330	9556782	7568305	5961124
Layer5	4586667	5505984	6727150	9514850	8077869	6018038
Layer6	3897014	4646656	5710189	5958755	3924313	4197026
μ (Mean)	4176928	4787883	6089157	8765451	7031285	5614427
σ (standard deviation)	244678	517912	352212	1274014	1432550	643191
$CV=\sigma/\mu$	5.86%	10.82%	5.78%	14.53%	20.37%	11.46%

Table 9: The distribution of layer areas of MFDPrior in ISPD98 benchmark

6.4.1 The Distribution of Layer Spaces

CV is the area coefficient of variation ($CV = \sigma/\mu$) and is allied to the given parameter F_{ub} and desired layer K . When the parameter F_{ub} or K is larger to permit more freedom area space of a chip, CV probably is larger. The average coefficient of variation depicts the distributions of chip areas that we defined before.

From Table 9, the average coefficient of variation of MFDPrior is 7.73%. This average coefficient of MFDPrior is 1.22X bigger than FDPrior and exhibits the distributions in MFDPrior are more uneven than in FDPrior algorithm. In cases ibm01 ~ ibm06, the average coefficient of variation is 5.14%. While F_{ub} is 12 and desired layer is 5, the average coefficient of variation is 5.57% in cases ibm07 ~ ibm12. And the average coefficient of variation is 11.47% in ibm13 ~ ibm18, when F_{ub} is 15 and desired layer is 6. Because MFDPrior operates the modified bottom-up approach in the partitioning phase and revokes the mapping cells into a layer phase, instead of mapping all cells in to a chip at a time. This more unbalance circumstances are unavoidable due to accomplish the bottom-up methodology and mapping whole cells once.

6.4.2 Compare Results with Multilevel of PP3D

Using multilevel methods is one of well-known approaches to solve the partitioning problem. In the partitioning filed, multilevel of FM algorithm is a celebrated algorithm [33]. Hence, we organize the multilevel structure of PP3D as the multilevel topology of traditional algorithm. Fig. 19 shows and compares experimental results with the multilevel of PP3D and MFDPrior algorithms.

Even though, our research accomplishes hierarchically multilevel structure to minimize the numbers of TSVs in MFDPrior. The advancements of MFDPrior are not as great as our exceptions, probably because the coarsening methods are not quite adequate under the uneven size of cells in a hypergraph. Therefore, applying which

type of coarsening methods is the significant topic in the multilevel techniques. Comparing with original PP3D that we have mentioned in Chapter 2, multilevel structure of PP3D earns superior solutions 3.25X in average. Fortunately, MFDPrior still obtains averagely 9.07X superior solution qualities than multilevel structure of PP3D.

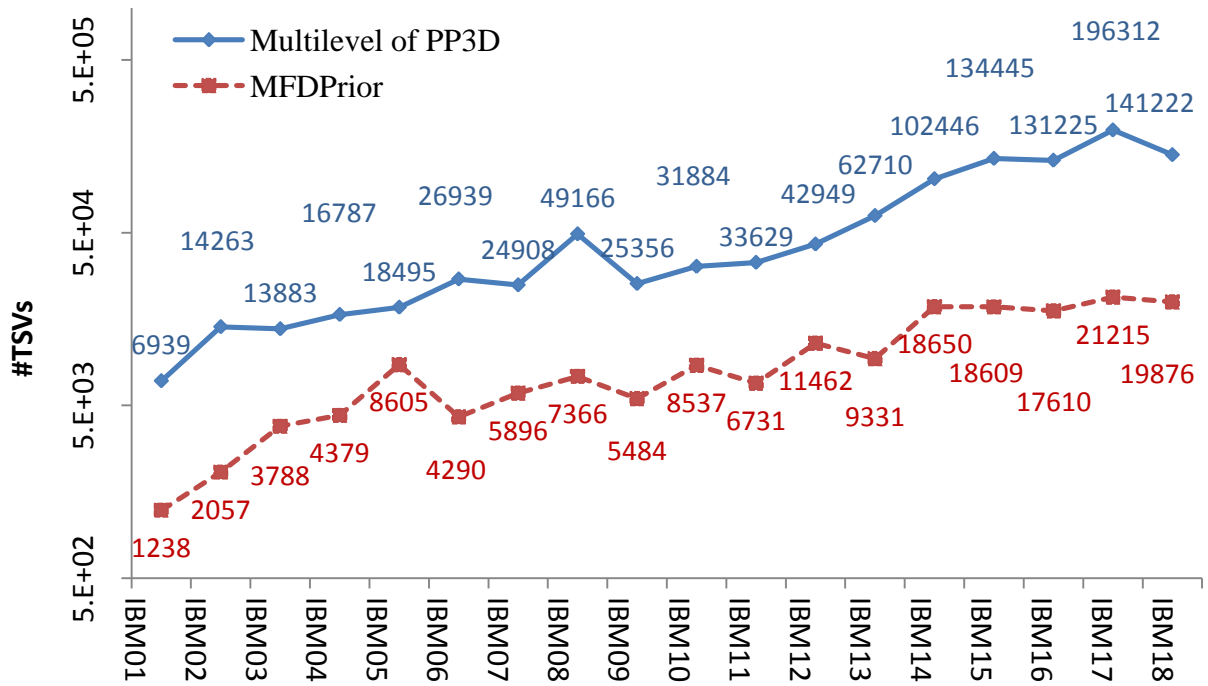


Figure 19: Comparison of average TSVs with Multilevel of PP3D

Chapter 7

Conclusion

The purpose of our research is providing a new field of vision by exploring the applications of N-body algorithms into EDA field. Since 3DICs technology has been considered as a solution to the challenges of large die area and long global wire delay, a study case is deliberated in the partitioning problem on a 3DIC.

This thesis proposes an innovative multilayer partitioning algorithm, FDPrior, for 3DICs. FDPrior is designed to take performance advantages of multi-core GPGPU systems and maintain the runtime scalability in the future large scaled 3DICs. In conclusion, FDPrior has three important strengths: 1) massive parallelism 2) better result quality by force-directed approach 3) bottom-up layer construction. The fundamental ideas are inspire a non-heuristic approach by the scalable force-directed concept based on N-body simulation. FDPrior merged these two methods and exposed the parallelism on GPGPU. Besides, FDPrior used the bottom-up prioritized layer construction to minimize synchronization overhead. On account of the bottom-up construction, the uneven distributions of layer area are inevitable. Compared with PP3D and conventional FM algorithms, FDPrior achieved 7.71X and 5.95X superior solution quality and attained 3.35X and 303.66X runtime speedup, respectively.

This thesis also focuses on enhancing FDPrior engine by adopting the multilevel scheme, called MFDPrior. MFDPrior obtains 1.46X superior solution quality and reaches 1.44X speedup as compares with former FDPrior algorithm.

Bibliography

- [1] J. F. Croix and S. P. Khatri, "Introduction to GPU programming for EDA," in Proc. ICCAD, pp. 276-280, 2009.
- [2] M. J. Stock and A. Gharakhani, "Toward efficient GPU-accelerated N-body simulations," AIAA paper, vol. 608, pp. 7-10, 2008.
- [3] Lars Nyland, Mark Harris, Jan Prins, Fast N-Body Simulation with CUDA, Nvidia GPU Gems 3, Chapter31.
- [4] S. Aarseth, Gravitational N-body simulations, Cambridge Univ. Press, 2003.
- [5] S. Das, A. Fan, K.-N. Chen, et al., "Technology, performance, and computer-aided design of three-dimensional integrated circuits," in Proc. of the international symposium on Physical design, pp. 108-115, 2004.
- [6] S. F. P. Zwart, R. G. Belleman, and P. M. Geldof, "High-performance direct gravitational N-body simulations on graphics processing units," New Astronomy, vol. 12, pp. 641-650, Nov 2007.
- [7] A. Marciniak, Numerical solutions of the N-body problem, D. Reidel, 1985.
- [8] R. Spurzem, "Direct N-body simulations," Journal of Computational and Applied Mathematics, vol. 109, pp. 407-432, 1999.
- [9] B. Marcos, T. Baertschiger, M. Joyce, et al., "Linear perturbative theory of the discrete cosmological N-body problem," in Physical Review D, vol. 73, p. 103507, 2006.
- [10] G. Contopoulos, N. Spyrou, and L. Vlahos, Galactic dynamics and N-body Simulations, vol. 433, 1994.

-
- [11] N. Arora, A. Shringarpure, and R. W. Vuduc, "Direct N-body Kernels for Multicore Platforms," in Proc. ICCP, pp. 379-387, 2009.
- [12] K. R. Meyer, Periodic solutions of the N-body problem, Springer, 1999.
- [13] Nvidia, CUDA C Best Practices Guide version 3.2
- [14] Nvidia, <http://www.nvidia.com>
- [15] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich, GORDIAN: VLSI placement by quadratic programming and slicing optimization, IEEE Trans. on Computer-Aided Design, 10(3), pp. 356–365, March 1991
- [16] X. Dong and Y. Xie, "System-Level Cost Analysis and Design Exploration for Three-Dimensional Integrated Circuits," in Proc. ASP-DAC, pp. 234-241, 2009.
- [17] B. Catanzaro, K. Keutzer, and S. Bor-Yiing, "Parallelizing CAD: A timely research agenda for EDA," in Proc. DAC, pp. 12-17, 2008.
- [18] T. Yan, Q. Dong, Y. Takashima, and Y. Kajitani, "How Does Partitioning Matter for 3D Floorplanning?" in Proc. GLSVLSI, pp. 73-78, Apr. 2006.
- [19] G. J. Wipfler, M. Wiesel, and D. A. Mlynski, "A Combined Force and Cut Algorithm for Hierarchical VLSI Layout," in Proc. DAC, pp. 671-677, 1982.
- [20] I. H. R. Jiang, "Generic integer linear programming formulation for 3D IC partitioning," in Proc. SOCC, pp. 321-324, 2009.
- [21] G.Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Application in VLSI Domain," IEEE Trans on Very Large Scale Integration Systems, 7(1), pp. 69-79, Mar. 1999.
- [22] C. Ferri, S. Reda, and R. I. Bahar, "Parametric Yield Management for 3D ICs:

-
- Models and Strategies for Improvement,” ACM Journal on Emerging Technologies in Computing Systems, 4(4), pp. 19:1-19:22, 2008.
- [23] W.-J. Chen, H.-K. Kuo, T.-H. Chiu, et al., "FDPrior: A force-directed based parallel partitioning algorithm for three dimensional integrated circuits on GPGPU," in Proc. VLSI-DAT, pp. 1-4. 2011.
- [24] Hsien-Kai Kuo, Bo-Cheng Charles Lai and Jing-Yang Jou,” Unleash the Parallelism of 3D IC Partitioning On GPGPU”, in Proc. SOCC, 2011.
- [25] C. M. Fiduccia and R. M. Mattheyses, “A Linear-Time Heuristic for Improving Network Partitions,” in Proc. DAC, pp. 175-181, 1982.
- [26] N. R. Quinn, “The placement problem as viewed from the physics of classical mechanics”, in Proc. DAC, pp. 173-178, June 1975.
- [27] L. T. Wang, Y. W. Chang, and K. T. Cheng, Electronic design automation: synthesis, verification, and test, Morgan Kaufmann, 2009.
- [28] Nvidia, CUDA programming guide version 2.3.1
- [29] ISPD-98, <http://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html>
- [30] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in Proc. Design Automation Conference(DAC),pp. 343-348, 1999.
- [31] G. Karypis, R. Aggarwal, V. Kumar, et al., "Multilevel Hypergraph Partitioning: Application In VLSI Domain," in Proc. DAC, pp. 526-529, 1997.
- [32] H. Yu Cheng, C. Yin Lin, and C. Mely Chen, "A multilevel multilayer partitioning algorithm for three dimensional integrated circuits," in Proc. Quality Electronic Design (ISQED), pp. 483-487, 2010.
- [33] hMetis, <http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download>