Electronic Design Automation with Graphic Processors: a Survey

Yangdong Deng

dengyd@tsinghua.edu.cn

Shuai Mu

mus04ster@gmail.com

Institute of Microelectronics

Tsinghua University

Beijing, 100084, China

Today's IC architects depend on Electronic Design Automation (EDA) software to conquer the overwhelming complexity of VLSI designs. As the complexity of IC chips is still fast increasing, it is critical to maintain the momentum towards growing productivity of EDA tools. On the other hand, single-core CPU performance is unlikely to see significant improvement in the near future. It is thus essential to develop highly efficient parallel algorithms and implementations for EDA applications so that their overall productivity can continue to increase in a scalable fashion. Among various emergent parallel platforms, Graphics Processing Units (GPUs) now offer the highest single-chip computing throughput. A large body of research, therefore, has been dedicated to accelerating EDA applications with GPUs. This paper is aimed to develop a timely review of the existing literature on GPU based EDA computing. Considering the substantial diversity of VLSI CAD algorithms, we extend a taxonomy of EDA computing patterns, which can be used as basic building blocks to construct complex EDA applications. GPU-based acceleration techniques for these patterns are then reviewed. On such a basis, we further survey recent works on building

efficient data-parallel algorithms and implementations to unleash the power of GPUs for EDA applications.

Categories and Subject Descriptors: J.6 **[Computer-Aided Engineering**] – Computer-aided design (CAD).

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Electronic Design Automation (EDA), VLSI, GPU, Graphics Processor, GPGPU, logic simulation, circuit simulation, matrix, linear algebra, sparse matrix, graph traversal, graph algorithm, dynamic programming, simulated annealing, structured grid

1. INTRODUCTION

As the foundation of information technology, Integrated Circuits (ICs) are playing a fundamental role in our society. In the foreseeable future, IC technology will still be one of the major enablers for sustainable development. To further improve the working efficiency and living standards of the human beings, the number of ICs deployed around the world will still be rapidly increasing in the future. It is predicted that 15X more transistors are going to be deployed in the next 5 years to "manage, store, and interpret data" [Otellini 2011].

At the same time, the complexity of ICs has been growing as indicated by the Moore's law to maintain the momentum towards increasing performance and functionality. Today it is already feasible to integrate over 7 billion transistors on a consumer IC chip [NVIDIA 2012a]. To conquer the overwhelming complexity of modern ICs, circuit designers depend on Electronic Design Automation (EDA) software to convert a design intention into working silicon. EDA tools, therefore, have to be scalable with the growing IC complexity so that the design turnaround time can be kept in a reasonable level. Current EDA tools are facing challenges from two ends, big system and small physics [Rutenbar 2007]. The former means the integration of a whole hardware/software system onto a single chip, while the latter involves the manufacturability, reliability and other issues incurred by the shrinking physical size of IC fabrication processes. Both trends pose significant requirements to the processing throughput of EDA software.

In the past, the performance scalability of EDA tools had always been the result of two interacting factors, smarter algorithms and faster CPUs. The latter factor is especially handy because the same EDA algorithm automatically runs faster on a CPU with higher performance. In early 2000s, however, single-core CPU performance is saturating due to the inability to extract more instruction level parallelism and improve power efficiency. Such a stall in computing performance had serious implications on the design turnaround time of IC design projects. Given the complexity of today's IC

designs, the run time of EDA applications can still be excessive even using the best algorithm to date. For instance, a timing analysis will take a couple of hours to perform on a 5M-gate design. Such a run time seriously constrain the number of optimization steps that can be conducted in a given design turnaround time, since virtually every post-synthesis optimization operation requires a run of timing analysis to validate the correctness. A run time of a few hours suggests that only a small portion of the complete solution space can be explored and the design quality has to be relaxed. Another example is the circuit simulation problem. Given a Giga-Hertz phase-lock loop (PLL) circuit, a transient analysis needs to simulate the circuit for millions of cycles before the frequency can be stabilized. Thus a complete run will take months to finish on a single CPU. Besides, the continuously shrinking market window of today's electronic appliances also poses challenging requirements to the productivity of EDA software.

In spite of the relative saturation of single-core CPU performance in the conceivable future, the semiconductor processes are still offering continuously growing integration capacity. As a result, all major CPU vendors switched to offer multi-core products since 2006. Multi-core processors are inevitably becoming the dominant computing platform for EDA applications. Accordingly, it is crucial to develop parallel solutions to EDA software such that the momentum of function increase in VLSI designs can be maintained [Catanzaro et al. 2008].

In the past a few years, major EDA vendors proposed R&D initiatives to take advantage of the computing power of multi-core processors [Sapatnekar et al. 2008]. At the present time, the POSIX threads or Pthreads [IEEE 2004] based multithreading has been the most popular programming model for multi-core CPUs. Multithreaded versions of cutting-edge EDA software have already been released. Such applications include parallel circuit simulator (e.g., [Cadence 2008], [Synopsys 2008a]), router (e.g., [Synopsys 2008b]) and physical verification (e.g., [Kapoor et al. 2004]). Among these, multithreaded parallel circuit simulation proves to be

especially successful. Meanwhile, the academia also introduced parallel algorithms for many EDA applications (e.g., [Hsu et al. 2008; Lu et al. 2009; Muller 2006; Sapatnekar et al. 2008]).

Despite their many successful applications, the multithreaded parallel programming model on multi-core CPUs still has serious limitations. A CPU thread is associated with a relatively high overhead in initialization, context switching, and synchronization [Butenhof 1997]. Accordingly, Pthreads and similar programming models belong to the category of coarse-grain multithreading, which suggests parallel processing of tasks and/or large chunks of data of a problem. However, many complex EDA applications feature abundant fine-grain parallelism (i.e., data parallelism) exemplified by matrix and graph operations. A multi-core microprocessor at most supports a few tens of threads and cannot fully take advantage of the inherent fine-grain parallelism. In addition, the scalability of a coarse-grained multithreaded program is seriously limited by the thread management overhead. A context switching of a thread on a multi-core CPU takes a few hundreds of micro-seconds [Li et al. 2007]. Generally, such an overhead will outweigh the speed-up of increasing parallelism when the number of threads is beyond a given level. A recent work showed that the performance of a highly optimized parallel logic simulator saturated at 15 threads on a 10-core CPU [Pingali et al. 2011].

The above problems of multi-core processors as well as the pursuit for more computing power motivate EDA researchers and engineers to explore alternative parallel computing platforms. Recently, Graphic Processing Units (GPUs) have emerged as a new general purpose computing platform [Owens et al. 2005; Blythe 2008; Owens et al. 2008]. GPUs were originally designed as application specific ICs for graphics rendering. Pushed by the relentless pursuit for better visual experiences, GPUs evolved to offer both high programmability and superior computing throughput. In 2004, NV35 GPU began to deliver a higher level of performance than the best CPU at that time. Current GPUs outperform their multi-core

CPU equivalents by a factor of over 30 in terms of peak computing throughput.

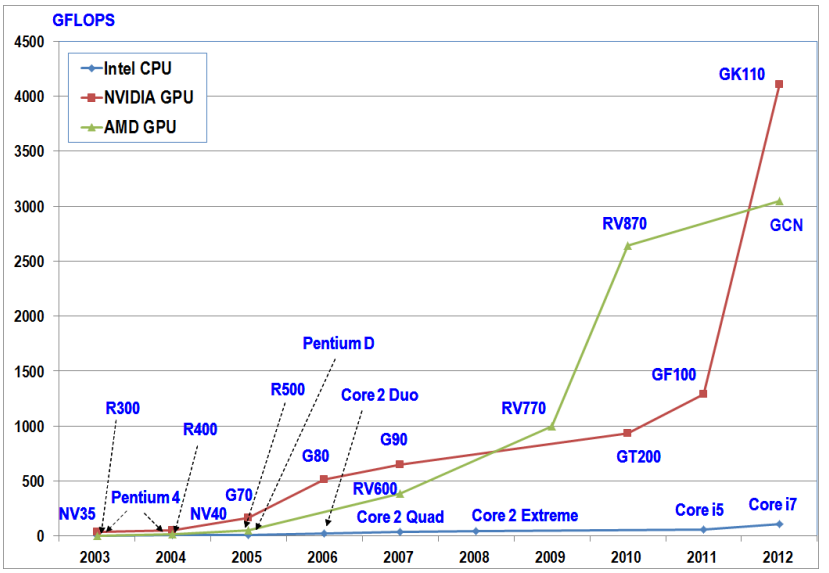


Fig. 1.1 Comparison of peak throughput of CPUs and GPUs

The above performance trend is depicted in Fig. 1.1, where the computing throughputs of NVIDIA and AMD GPUs and Intel CPUs are compared in terms of Giga FLoating Operation Per Second (GFLOPS). We collected performance data from publically available datasheets ([NVIDIA 2012b; AMD 2012]). GPU chip makers usually release multiple GPUs with varying performance levels at each technology node. Meanwhile, the above 3 companies have different schedules for releasing new products. In Fig. 1.1 we only show the "flagship" GPU for each generation and take NVIDIA's release schedule as the time reference. Clearly, GPU has been outperforming CPU since 2004 and the performance gap is still broadening.

Along with the high computing throughput, GPUs are also equipped with a high bandwidth memory bus because it is installed on the graphics card and dedicated to GPU applications. The memory characteristics of major GPUs are demonstrated in Fig. 1.2. The bandwidth values of 4 generations of DDR memories, i.e. the memory standard for CPUs, are also depicted as reference. The latest NVIDIA and

AMD GPUs have a peak memory bandwidth of 208 GB/s and 264 GB/s, respectively, while the current DDR3 memory standard only supports 17GB/s (the next generation DDR4 will double the bandwidth to 34GB/s) [JEDEC 2012]. Certainly the superior memory bandwidth of GPUs will significantly benefits memory intensive EDA applications.

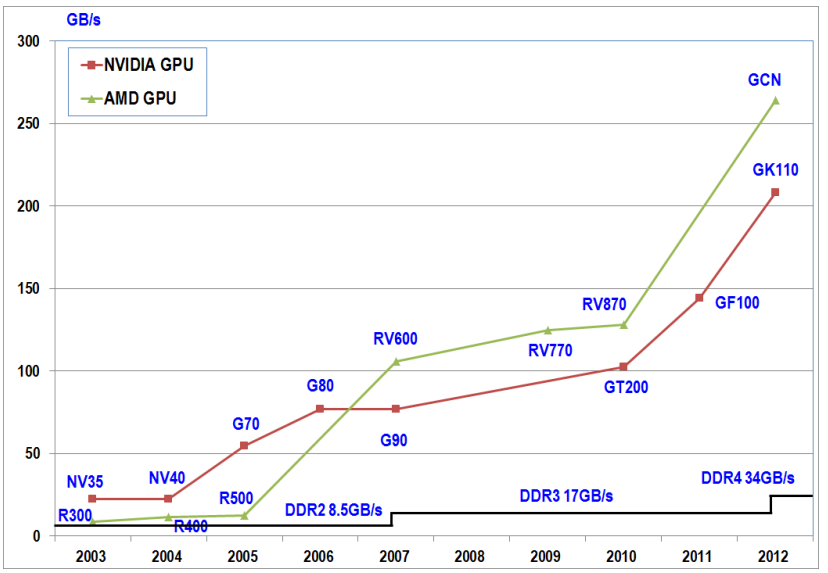


Fig. 1.2 Comparison of peak throughput of CPUs and GPUs

Traditionally, GPUs are programmed with shading languages like OpenGL [OpenGL 2012]. Although OpenGL can be used for general purpose computing on GPUs (GPGPU), the resultant programming process is laborious and error-prone. To ease the programming effort of GPGPU, NVIDIA introduced the Compute Unified Device Architecture (CUDA) technology [NVIDIA 2007a; Nickolls et al. 2008] so that programmers can develop GPGPU programs in a C/C++ alike language with a few extensions. While CUDA can only be used on NVIDIA GPUs, OpenCL is defined by a group of industry players as a standard cross-platform GPGPU language **[**Khronos 2012**]**.

The synergy of GPU hardware and software has resulted in successful applications in a diverse range of scientific and engineering domains [Blythe 2008; Owens et al. 2008]. On workloads with appropriate computing and memory accessing

patterns, GPU can even attain a speed-up of over 100X. It is thus appealing to unleash the computing power of GPU for EDA applications.

Different CPUs, GPUs adopt a fine-grain multithreading model. Equipped with dedicated hardware for context switching, GPU threads are light-weighted and excel in massively data-parallel processing. Such an execution model makes GPU proper for EDA applications featuring data-parallelism. There is already a large body of literature presenting encouraging results on utilizing GPU to solve various EDA problems. GPGPU proved to be effective in such time consuming applications as system level design, logic simulation, timing analysis, power grid analysis, placement, and routing. The positive results suggest that the superior computing power of GPUs can be unleashed by developing carefully designed data-parallel algorithms and highly tuned implementations.

On the other hand, EDA applications pose unique challenges to the GPGPU model. The nature of circuits determines that the underlying data structures capturing IC designs tend to be irregular. Typical EDA applications are thus constructed on the basis of such irregular data structures as sparse matrix, tree, and graph[1]. The resultant memory accessing patterns are less amenable to GPUs, which only have a limited capacity of cache and assume regular memory accesses to fully utilize its large memory bandwidth. Accordingly, current works on GPU based EDA computing generally resort to 2 strategies: 1) identifying regular sub-problems in an EDA application and then use GPU as an accelerator for them; and 2) re-designing or re-structuring algorithms on GPU so as to convert irregular data accesses into (at least partially) regular ones. Another challenge is that EDA applications are extremely complex and cover many different domains of computations. Accordingly, an application-by-application parallelization approach can be infeasible. A viable line of attack, instead, is to identify the

[1]There exist special cases where the data structure can be quite regular. One such typical example is the power distribution network, which in many designs consists of a relatively regular power mesh.

fundamental computing patterns and perform parallelization on them. Such a pattern based strategy of parallel programming proves to be crucial for many other software applications [Mattson et al. 2004].

In this paper, we present an up-to-date survey on the progresses in GPU accelerated EDA computing. Considering the high complexity of EDA applications, an essential objective of this work to extract key computing patterns of EDA and present state-of-the-art GPU programming techniques to resolve such patterns. We believe this approach will substantially ease the deployment of GPUs in future EDA software. This paper focuses on using GPU to accelerate applications in the EDA domain, while the techniques also have wide applications in many other scientific and engineering domains. Interested readers please also refer to [Owens et al. 2007; Blythe 2008; Owens et al. 2008] for surveys on applications in other disciplines.

The remainder of this paper is organized as follows. Chapter 2 provides an overview of GPU hardware architectures and the corresponding data-parallel programming model. In Chapters 3 and 4, we develop a taxonomy for the basic computing patterns of EDA applications and then review relevant GPU programming techniques for these patterns. In Chapter5, we survey successful applications of GPU accelerated EDA computing. In Chapter6, we conclude this work and propose future research directions.

2. GPU ARCHITECTURE AND PROGRAMMING MODEL

In this chapter, we present a high-level overview on the GPU architecture and the corresponding data-parallel programming model. The organization of this chapter is as follows. First we briefly introduce the history of GPUs. Then the hardware architectures of various GPUs are reviewed in section 2.2. Section 2.3 uses NVIDIA CUDA to exemplify the GPGPU programming model. The program execution model and its impact on program design are discussed along with the programming model.

2.1 GPU Background

A GPU is an application specific processor originally designed for 3-D graphics rendering. Although its history can be dated back to early 1980s (e.g., [Clark 1982; Cook et al. 1987; Levinthal and Porter 1984]), GPU had not entered the mass market until 1990s, when 3-D real-time games became popular [Fatahalian and Houston 2008; Blythe 2008]. Nowadays, GPUs are pervasively installed in computers/servers, game consoles, tablets, and smart phones. In high-end computing platforms, a GPU is usually mounted on a graphic card, which is plugged onto a motherboard via a 16-lane PCI Express bus [PCI-SIG 2002]. Fig. 2.1 illustrates the diagram of a computer with graphic card. In middle- and low-end machines, a GPU is directly assembled on a motherboard. The current GPU market is mainly dominated by three companies, among which

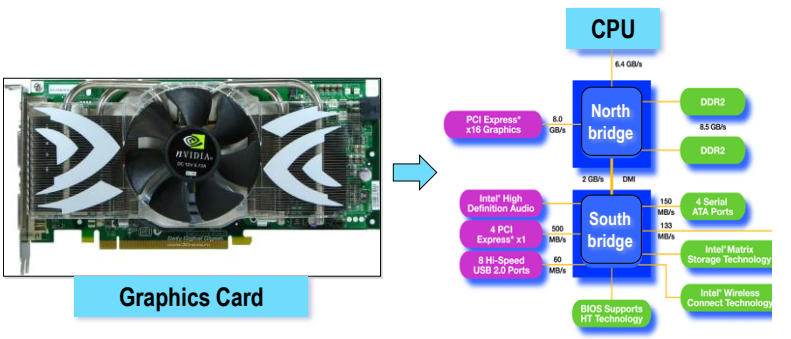


Fig.2.1 Diagram of a computer with a 3-D graphics card

NVIDIA and AMD are the largest two high-end GPU providers, and Intel is the largest manufacturer of medium- and low-end GPUs [McGlaun 2009]. In addition, Intel had attempted to move into the high-end market through its effort on the Larrabee GPU [Seiler et al. 2008].

During the process of graphics rendering, a CPU is responsible of creating 3-D scenes, decomposing the 3-D objects into geometric primitives (usually polygons or triangles in a 3-D space), and transferring the primitives to GPU through a communication fabric. Upon receiving the geometric primitives, a GPU processes them through a graphics pipeline consists of geometry, rasterization, fragment processing, and pixel operations [Fatahalian and Houston2008]. The geometry processing stage projects the input 3-D geometric objects into a 2-D visual plane (i.e., corresponding to a 2-D display). During this stage, a GPU only performs computations on the vertices of the primitives and so the stage is often designated as vertex processing. At the rasterization stage, a spatial sampling is carried out so that the 2-D geometric primitives after vertex processing can be converted into fragments. Although having the same size, a fragment differs from a pixel because each primitive has its own set of fragments. The fragment processing stage determines the RGB values of every fragment. The resultant fragments are checked for visibility since multiple fragments may correspond to an identical pixel. The visible fragments are then written into a frame buffer in the video memory for final display.

Real-time 3-D graphics rendering has always been one of the most computing-intensive applications. Today a typical 3-D scene consists of ~1M triangles, equivalent to ~3M vertices and ~25M fragments. To deliver satisfying visual effects, at least 30 frames have to be rendered in one second. The above figures suggest a throughput of 90M vertices and 750M fragments per second. Since multiple floating-number or integer operations will be performed on each vertex and fragment, the total computing demand is substantial. Fortunately, the rendering process is inherently parallel. Generally, the operations of vertices or fragments in a frame

are independent. Hence GPU architects deploy multiple identical processing elements on one chip to handle different vertices (fragments). Because all vertices (fragments) are handled by the same program, the GPU execution naturally follows a single instruction, multiple data (SIMD) model. Such a highly parallel architecture allows GPU performance to rapidly boost to a higher level than what can be achieved on CPUs. In fact, GPUs have been able to deliver a higher throughput than CPUs since 2004. Fig. 1.1 already compares the peak throughput between modern CPUs and GPUs.

As the pixels of every frame need to be written into video memory, 3-D graphics rendering is a memory intensive application. However, the memory access patterns of graphics processing significantly differ from those of general CPU computing. On the one hand, the spatial locality of graphics operations is totally predictable because each time a complete video frame has to be written to memory. On the other hand, there is almost no temporal locality because we generally do not need to use the data again after rendering them. Accordingly, GPUs adopt a very different memory design philosophy. First of all, GPU communicates with video memory through a dedicated bus since both the GPU and the memory chips are installed on a single graphic card. Such a configuration has important implications on the peak memory bandwidth. First, GPU has a 384- or 512-bit bus to off-chip memory. Such a width is much wider than that of the memory bus of a typical CPU. Second, the GPU memory bus is dedicated to graphics and cannot be occupied by other traffic. Third, GPUs are capable of coalescing multiple memory requests that follow certain patterns into a single, blocked one. The coalesced request contains all required data and can be made available in a single memory access. Obviously, such a feature is designed to match the behaviors of continuously reading/writing the contents of a frame. The GPU memory philosophy enables a superior memory bandwidth, which also began to outperform CPU bandwidth in 2004. Today GPU memory bandwidth is higher than its CPU counterpart by over 10 fold. Meanwhile, traditionally GPUs are only equipped with a relatively limited capacity of cache because the

predictability of memory behaviors mitigates the pressure of hiding memory latency. Instead, the silicon estate of GPU chips is mostly dedicated to computing resources.

The superior performance of GPUs attracted researchers to explore their applications in non-graphics domains. The concept of GPGPU was born in early 2000s. An excellent review on the early work of GPGPU can be found in [Owens et al. 2005]. In the initial stage of GPGPU, the algorithm for a non-graphics problem had to be implemented with a graphics language such as OpenGL, while the data structures must be translated into graphics objects. Before the introduction of NVIDIA G80, GPUs used separate hardware resources for vertex and fragment processors, which were equipped with different programming mechanisms. Thus a GPGPU program was then executed in parallel by either vertex processors or fragment processors (most GPGPU works used fragment processors). Such an execution model obviously led to waste of hardware resources. Meanwhile, the lack of general-purpose software development tools also made GPGPU a challenging job, especially for programmer with little graphics experience.

With the release of G80 GPU, NVIDIA made fundamental changes on the GPU architecture [NVIDIA 2006]. A key renovation was that a general processor replaced dedicated vertex and fragment processors. A G80 GPU integrates 128 identical 32-bit processors and all of them are programmed in a unified manner. Along with G80, NVIDIA introduced a new programming model for general purpose programming as well as the corresponding software development toolkit. The combined hardware and software solutions constitute a platform technology, which is designated by NVIDIA as Compute Unified Device Architecture, or CUDA [NVIDIA 2007a]. CUDA uses a single program multiple data (SPMD) programming model in which multiple threads concurrently work on difference pieces of data. In roughly the same timeframe, AMD introduced their GPGPU products and software [AMD 2009]. Meanwhile, Intel also introduced a many-core processor, Larrabee [Seiler et al. 2008], for general

purpose computing[2]. The corresponding software development kit, Ct, was released in 2007 [Ghuloum et al. 2007]. Programmers can develop Ct programs and then emulate their execution on current Intel CPUs with SIMD support. The above 3 frameworks are incompatible at either source code or binary level. So there is no way but to porting a program from one GPU to another by re-writing the code. To provide a cross-platform GPU programming environment, Khronos Group, which is an industry organization comprised of leading industry players, announced OpenCL as an open, royalty-free GPGPU standard [Khronos 2012]. Currently, NVIDIA, AMD, Intel and many GPU vendors already support OpenCL on their GPUs.

2.2 GPU Hardware Architecture

In this section, we review representative GPU microarchitectures that support general purpose computing. We start with NVIDIA Fermi GPU (also codenamed as GF100). Although Fermi GPU is not the latest NVIDIA GPU yet, it was the first generation of GPU specifically designed with GPGPU capability [Wittenbrink et al. 2011]. So we use it as a reference microarchitecture to explain GPU hardware. On such a basis, we further introduce Fermi GPU's successor, Kepler GPU, as well as AMD GPUs with an emphasis on the Graphics Core Next (GCN) microarchitecture [AMD 2011]. For completeness, we also briefly review Intel's Larrabee GPU [Seiler et al. 2008].

2.2.1 NVIDIA GPU

As the largest independent GPU provider, NVIDIA has already released 4 generations of GPUs with GPGPU capabilities. These GPUs support the same CUDA programming model, but with increasing performance and functionality. NVIDIA GPU chips are shipped in 3 different product lines. Among these, the Geforce series graphics cards are for gaming, Quadro series are

[2]Larrabee was originally designed as a GPU, but later re-positioned as a throughput processor and then merged into Intel's MIC microprocessor.

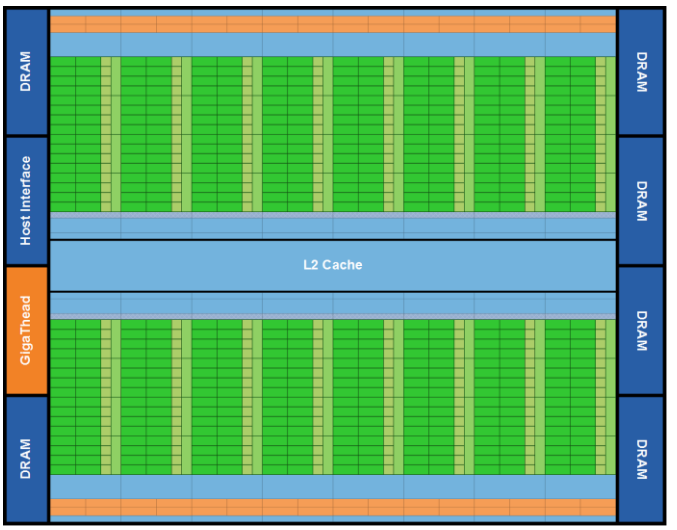


Fig. 2.2 Architecture of NVIDIA Fermi GPU [NVIDIA 2009]

for high-performance graphics applications, and Tesla series are acceleration cards for general purpose computing.

NVIDIA began to support CUDA based general purpose computing at the G80 generation, but Fermi GPU was the one designed by simultaneously considering graphics and GPGPU [NVIDIA 2009]. The microarchitecture of Fermi GPU is illustrated in Fig.2.2. The computing resources are distributed among 512 cores (also coined as CUDA cores, streaming processors or SPs), which are evenly distributed into 16 streaming multiprocessors (SM) or multiprocessors. Fermi also features a 768KB unified level-2 (L2) cache, which is specifically design for GPGPU applications. There are 6 memory controllers installed on Fermi and the aggregated off-chip memory bandwidth can be over 100GB/s.

A streaming multiprocessor is an independent instruction execution unit[3]. The internal structure of an SM is illustrated in

[3]The term, streaming multiprocessor, in NVIDIA's terminology is a bit misleading. First, "streaming" actually means the processing of streams of graphics primitives. Such a feature is less relevant for GPGPU.

Fig. 2.3. The arithmetic and logic instructions are performed by 32 CUDA cores, which are often designated as 32 lanes because these CUDA cores work simultaneously in a SIMD manner. Each CUDA core consists of a 32-bit floating-point multiply-add unit and a 32-bit integer ALU. Two neighboring CUDA cores can work together as a double precision floating-point unit. CUDA cores collect their inputs from and write their outputs to a multi-port, multi-bank register file. A multiprocessor is also equipped with 4 special functional units (SFUs). Each SFU consists of 4 multipliers as well as logic for transcendental functions, interpolations, triangular functions, and others. In addition, an SM has 16 load/store (LD/ST) units for address computation. Note that the CUDA cores, SFUs, and LD/ST units can be scheduled as 3 different groups in parallel.

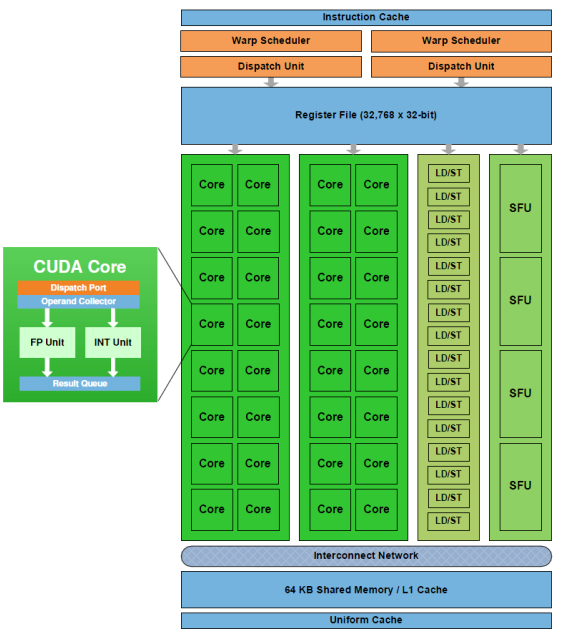


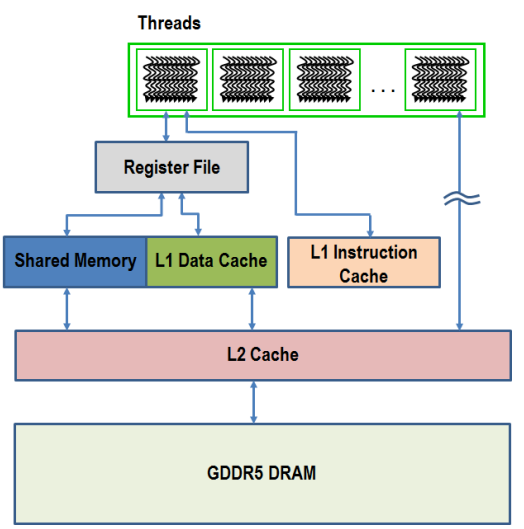
Fig. 2.3 Architecture a streaming multiprocessor [NVIDIA. 2009]

Second, a streaming multiprocessor only has a single instruction fetch unit, while a multiprocessor usually means it has multiple cores that independently fetch, decode and execution instructions. In this work, we use streaming multiprocessor and multiprocessor in an interchangeable manner.

A function to be executed on GPU is designated as a kernel, which usually consists of a large number of threads. A given number is organized as a warp in NVIDIA's terminology or a wavefront in OpenCL terminology[4]. A warp is the basic unit of GPU program execution. The threads in a warp always execute the same instruction. On Fermi GPU, a warp has 32 threads that are mapped to the 32 CUDA cores.

In each instruction cycle, a multiprocessor fetches instructions from its instruction cache and stores them into a buffer. Then the 2 warp schedulers in the multiprocessor can dispatch 2 instructions from different warps simultaneously. The 2 instructions are issued to any 2 of the 3 groups of executing hardware (i.e., CUDA cores, LD/ST units, and SFUs) in a multiprocessor.

Under ideal conditions, the 512 CUDA cores work together and thus a throughput of 1288 GFLOPS is attainable. Few real world applications, however, exhibit such ideal computing patterns. On the other hand, the peak performance does provide an important guideline for program optimization. One can use the gap between the peak and effective throughput to drive the performance tuning process.



[4]In this work, warp and wavefront are used in an exchangeable fashion.

Fig. 2.4 GPU memory hierarchy

Fig. 2.4 illustrates the memory hierarchy of Fermi GPU. The first layer of memory hierarchy is actually a register file located inside a multiprocessor. Besides load and store instructions, all operands during the execution of a GPU program are from the register file. An SM of Fermi GPU has 32,768 32-bit registers organized as a multi-port register file.

Each multiprocessor is equipped with a 64KB memory, which is decomposed into 2 parts, shared memory[5] and level-1 (L1) cache. The memory space can be configured as 16KB shared memory plus 48KB L1 cache, or vice versa. The shared memory is organized as 32 banks[6] with a 4-byte word as the basic storage unit. The addresses are assigned in an interleaved manner, i.e., address 0 in bank 0, address 1 in bank 1, until address 32 in bank 0 again. Each bank is capable of providing one a 4-byte word in one clock cycle. Ideally, when every thread in a group of 16 threads accesses data from a different bank, the peak bandwidth can be realized as $16 \times 4 = 64$ bytes/cycle. If such a condition is not satisfied, the so-called bank conflict happens. For instance, 2 cycles are required before two threads can receive their data if both access data from two distinguished addresses in the same bank. Such an access pattern is called a two-way conflict. Obviously, the worst case is a 32-way bank conflict[7], which takes 32 cycles to have data ready for all threads.

The shared memory serves as a scratchpad memory for 2 purposes. The first is to use it as a software-controlled cache so that frequently used data can be stored closer to the computing resource without suffering the long latency of global memory accesses. It differs from CPU cache in the sense that

[5]The term, shared memory, is also misleading. Shared memory usually means a memory area that can be accessed by multiple processor cores, but it stands for a memory section inside an SM and can be shared by certain threads in NVIDIA's terminology.

[6] The number of banks is 16 in GPUs prior to Fermi.

[7]If all threads access exactly the same address in a certain bank, then a broadcasting mechanism allows the data to be available to all threads in one cycle.

programmers have to take care of the loading and replacement of the contents of the shared memory. In addition, the shared memory is often used by programmers as a buffer to reorder the memory accesses to achieve memory coalescing (will be detailed later in this section).

Besides the on-chip memory, a GPU is typically attached with an off-chip DRAM memory, or so-called device memory through a dedicated memory bus. The Fermi GPU supports GDDR5 high performance DRAM [JEDEC 2009]. GDDR5 is optimized for bandwidth, but the latency of accessing the off-GPU memory can be as costly as 400~800 cycles (GPU core clock). The off-chip memory is logically partitioned into three areas, constant memory, texture memory, and global memory. The constant memory can be used to store invariable data during program execution. The texture memory is mainly for graphics applications. It is supported by dedicated texture cache, but has to be accessed with special functions. Most program data are to be stored in the global memory.[8]

To reduce the impact of long memory latency, today's GPUs are commonly enhanced with a memory coalescing mechanism. On G80 and G90 GPUs (i.e., compute capabilities 1.0 and 1.1 defined by NVIDIA), coalescing is attained when a warp of threads access 16 adjacent memory addresses (can be 4-byte, 8-byte, or 16-byte data objects). Note that the memory addresses required by these threads must differ. Then these memory accesses can be combined into a single operation. Such a blocked accessing mechanism significantly boosts the effective memory bandwidth by taking advantage of the parallel architectures of memories. In case of coalesced memory operation, only one memory latency need to be experienced before all data are available. Otherwise, the 16 memory accesses take 16 latencies. The original coalescing rules are rather restricted and cannot be attainable by many common computing patterns. On GF100 GPU, the coalescing

[8] Again the use of "global memory" in NVIDIA terminology is misleading because it actually means the memory that can be accessed by all threads. The equivalent term, device memory, is more accurate.

rules are significantly relaxed. Now the memory space is organized into segments. One segment has a length of 32 bytes for an 8-bit data access, 64 bytes for a 16-bit access, and 128 bytes for a 32-bit or 64-bit access. Any access patterns by a warp of threads can be coalesced into one or more memory operations, with each operation serves a unique memory segment.

2.2.3 NVIDIA Kepler GPU

Recently NVIDIA released a new generation of GPU, codenamed as Kepler [NVIDIA 2012a]. The overall organization of Kepler GPU is similar to Fermi, but the structure of streaming multiprocessor is overhauled as shown in Fig. 2.5.

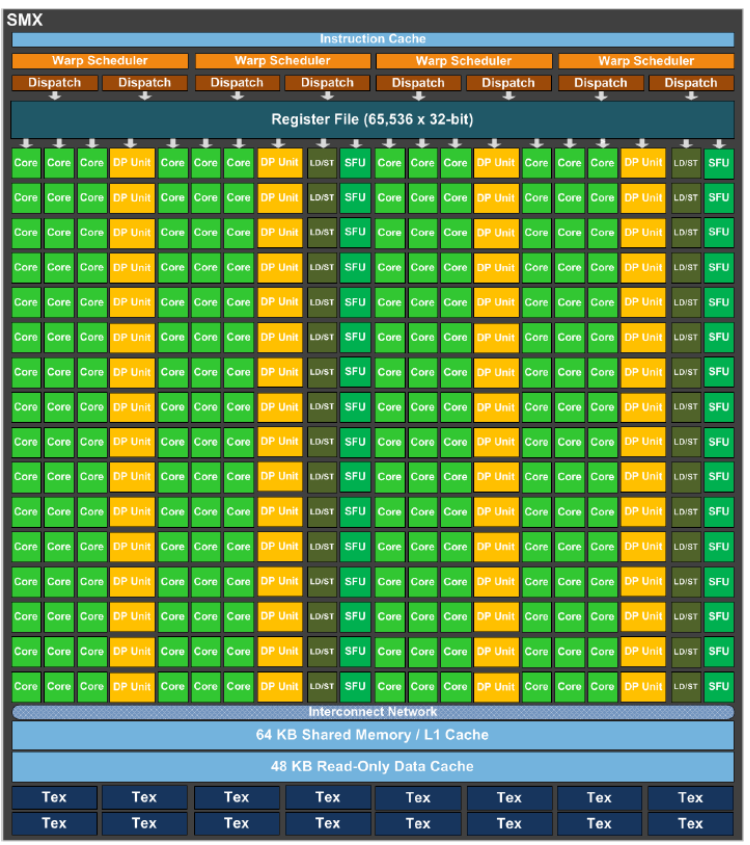


Fig. 2.5 Architecture of NVIDIA's new generation of streaming multiprocessor (SMX) [NVIDIA. 2012a]

The new streaming multiprocessor microarchitecture is designated as SMX to distinguish from the previous generations. Now an SMX has 192 single-precision CUDA cores plus 64 dedicated double-precision floating-point units. As a result, double-precision and single-precision floating point instructions can be scheduled in one single cycle. The number of load/sore units is doubled to 32, while the number of SFUs is increased to 32. The significant enhancement of SMX functionality is made possible by reducing the number of multiprocessors on Kepler GPU and migrating the dependency-checking logic to software.

With the significantly increased computing resources, now an SMX allows four warps of 32 threads physically running together. An SMX has 4 warp schedulers to coordinate the 4 warps. In addition, each warp scheduler has 2 instruction dispatch units. This allows better utilization of instruction level parallelism (ILP) because 2 independent instructions of a warp can be dispatched in one cycle.

Another new feature of SMX is the introduction of an interconnection network among different cores. It allows threads in a warp to directly exchange data via shuffle instructions. An illustration of the shuffle instructions is in Fig. 2.6. Such a feature has strong potential for FFT-alike applications that are abundant with regular data exchanges [NVIDIA 2012a]. It is actually appealing to study data-shuffling based algorithms [Ben-Asher et al. 1989] so that the power of the shuffling capability of Kepler GPU can be fully exploited.

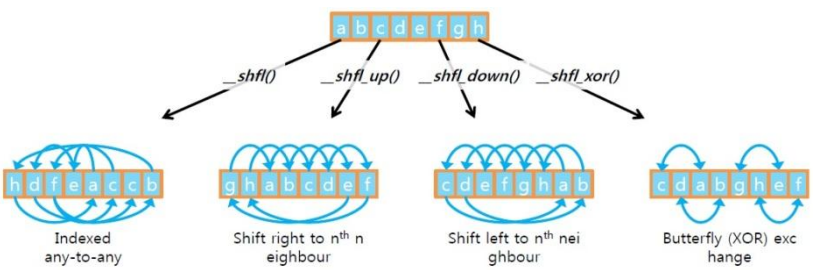


Fig. 2.6 Data shuffle instructions of Kepler GPU [NVIDIA. 2012a]

Perhaps the most revolutionary feature of Kepler GPU is the support of dynamic parallelism. In the past, only CPU can launch kernels on GPU. In other words, GPU has to return the control to CPU when finishing executing a kernel. This mechanism is illustrated in the left part of Fig. 2.7. As shown in the right side of Fig. 2.7, now the Kepler GPU allows a kernel to directly start another kernel without resorting to CPU. Obviously, such dynamic parallelism substantially relaxes the limitations of GPU programs by supporting nest parallelism [Blelloch 1990]. This feature has significant potential for EDA algorithms, where dynamic data and control flows are pervasive [Sherwani 1998].

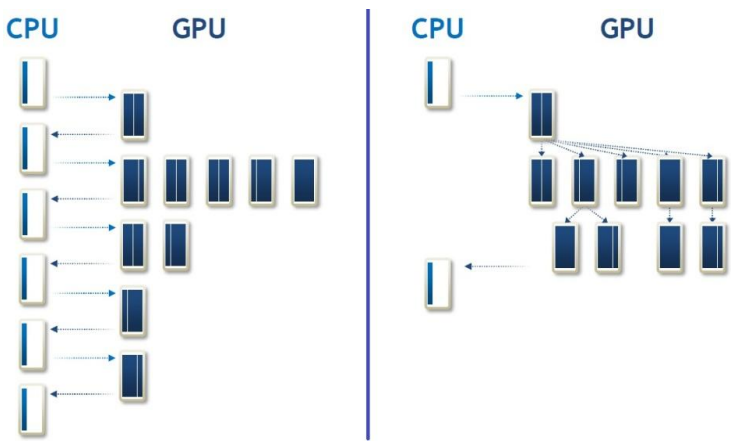


Fig. 2.7 Dynamic parallelism of Kepler GPU [NVIDIA. 2012a]

2.2.3 AMD GPU

AMD has been another major GPU provider, especially after its purchase of ATI in 2006. AMD had been building their GPU architecture by following the concept of very long instruction word (VLIW) [Fisher 1983; Smalley 2008a]. One such typical GPU is RV770, which consists of 10 SIMD cores, with each core being roughly equivalent to an SM in NVIDIA terminology. Every SIMD core is composed of 16 threaded processors. A threaded processor is actually a five-way VLIW processor equipped with 5 stream cores as basic functional units. Among the 5 cores, 4 cores are identical and designated

as *w*, *x*, *y*, and *z* ALUs. The remaining core is for transcendental functions. Such an architecture is designed to simultaneously support instruction level parallelism[9] (ILP) and thread level parallelism[10] (TLP). When a SIMD core process a group (a wavefront in AMD's terminology) of parallel threads, up to 5 independent instructions in a program can be scheduled and encoded into a single long instruction word for parallel execution by the 5 cores in a threaded processor. The compiler is responsible for extracting and scheduling the parallelism. On the other hand, NVIDIA GPUs mainly focus on thread level parallelism. The functional units inside a CUDA core cannot work in parallel. The VLIW architecture is more oriented to the graphics applications because parallelism can be extracted with relatively less effort. However, it is much more challenging to handle general purpose computing applications with such a static compilation based approach. The parallel hardware resources can be substantially underutilized due to the limitation of compiler [AMD 2011].

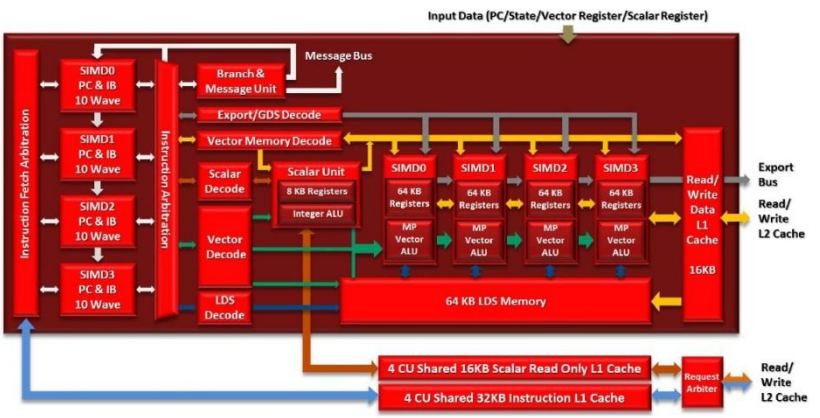


Fig. 2.8 Microarchitecture of GCN's compute unit [AMD 2011]

[9] ILP is the parallelism available in an instruction stream. If a given number of contiguous instructions are independent, then then can be execute in parallel by several hardware modules.

[10] TLP means the parallelism made available by running multiple threads in parallel. Instructions from different threads can be executed simultaneously by several hardware units.

To efficiently support both graphics and GPGPU workloads, AMD announced a new generation of GPU microarchitecture, Graphics Core Next (GCN) [AMD 2011]. The GCN microarchitecture is built around Compute Unit (CU), which is roughly equal to the streaming multiprocessor in NVIDIA's GPUs. The internal structure of a CU is illustrated in Fig. 2.8. A CU contains 4 SIMD units, each equipped with their own program counters and instructing fetching and decoding logic. A SIMD unit has an instruction buffer for 10 wavefronts of threads. The instruction scheduling logic always selects an instruction from 10 different wavefronts to dispatch (i.e., the 10 candidate instructions must be from dissimilar wavefronts). The frontend units are supported by a 32KB L1 instruction cache inside a CU. All L1 caches are further backed by a unified L2 cache.

The execution resources of a CU consist of both scalar and vector logics. Separate register files are installed for these two parts. The scalar logic is mainly for control flow operation. It consists of 2 pipelines, one for conditional branches and the other for address generation. The vector logic provides the major computing power of CU. Each SIMD unit has a 16-lane vector pipeline as well as a 64-lane register files of 32-bit registers. For a wavefront size of 64 threads, the SIMD unit needs 4 cycles to execute one instruction. Note that the 16-lane logic is for single-precision floating point operations, and so the double-precision and integer instructions have to be executed at a lower rate.

A CU also has a local data share (LDS) component serving as the scratchpad memory. Its internal structure is depicted in Fig. 2.9. LDS has a capacity of 64KB of 32-bit words. The memory address space is distributed into 32 banks, which are connected with an all-to-all crossbar interconnect fabric for fast data switching. LDS is shared by all 4 SIMD units. It has coalescing logic to coalesce memory requests from 2 different SIMD lanes.

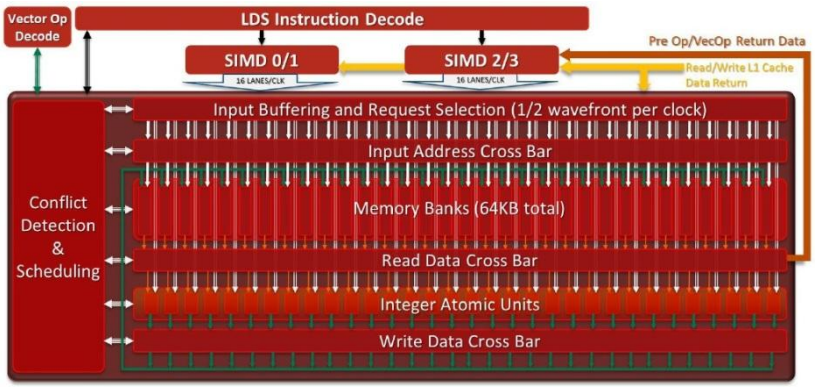


Fig. 2.9 Local data share of computing unit [AMD 2011]

The memory hierarchy of GCN is quite different from NIVIDIA GPUs. Fig. 2.10 shows the overall memory structure of GCN. First, a compute unit has its private 16KB L1 vector data cache. Every 4 CUs share a 32KB L1 instruction cache and a 16KB L1 scalar cache. The L1 cache blocks are connected via a crossbar to a distributed L2 cache, which consists of multiple 64KB or 128KB blocks. The L2 cache is globally coherent, while L1 cache is coherent across a computing unit. GCN's cache hierarchy is designed to be integrated with x86 CPUs.

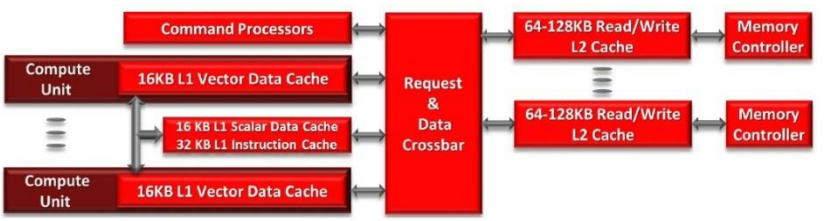


Fig. 2.10 Memory hierarchy of GCN [AMD 2011]

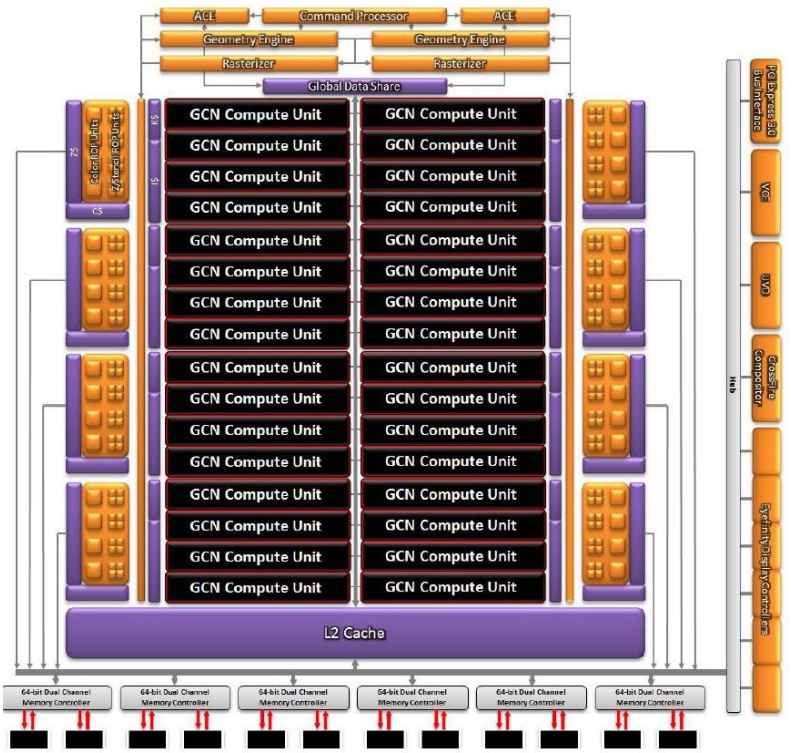


Fig. 2.11 AMD Radeon™ HD 7970 GPU [AMD 2011]

Compute units and distributed L2 cache blocks can be integrated in varying configurations to constitute different GPUs. Fig. 2.11 is the overall organization of AMD Radeon HD 7970 GPU.

2.2.4 Intel Larrabee GPU

Intel has long been the leading supplier for middle- and low-end integrated GPUs [Shimpi and Wilson 2008]. As Intel's first attempt to enter the high-end graphics and GPGPU markets, Larrabee is built on the basis of Intel's x86 architecture. The fundamental architecture of Larrabee is a group of in-order x86 cores interconnected by a bi-directional ring bus. All these cores have their own level-1 instruction and data caches, but share a common level-2 cache. Fig. 2.12 shows the overall organization of Larrabee.

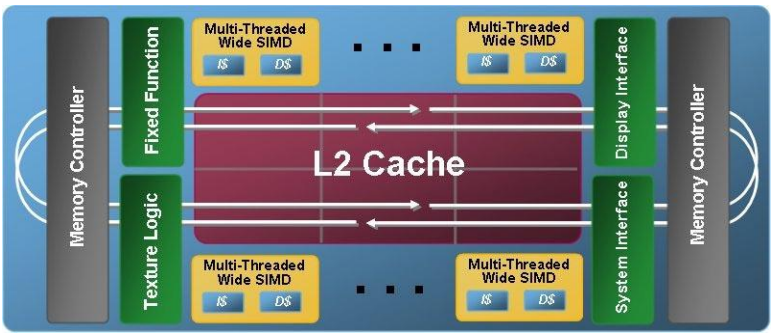


Fig. 2.12 Architecture of Intel Larrabee processor (adapted from [Folley2008])

As shown in Fig. 2.13, a Larrabee core has a rather conventional microarchitecture of x86 CPUs. It consists of both scalar and vector pipelines. The vector unit is actually a 16-lane vector ALU that is able to process 16 32-bit floating point operations in parallel. It has instruction fetch and decode logic as well as a 32KB L1 instruction cache and a 32KB L1 data cache. A 256KB L2 cache is attached to each core. Such a cache capacity certainly benefits the processing of large, unstructured data sets. In fact, the Larrabee GPU is based on a design philosophy different significantly from NVIDIA and AMD GPUs do. The former is closer to CPU to use cache to hide latency, while the latter two spend most silicon estate on computing hardware.

Larrabee takes a less revolutionary microarchitecture. It mainly depends on the compiler and the parallel programming model to extract the parallelism. It turned out to be a transitional product and the technology was later merged into Intel MIC many-core processor [Intel 2012].

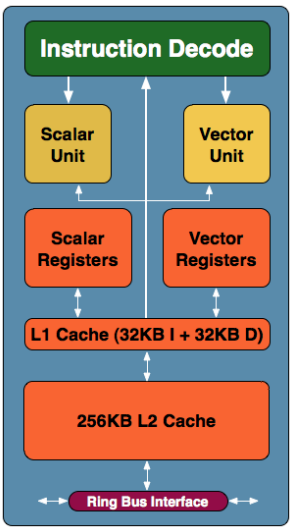


Fig. 2.13 Microarchitecture of Larrabee's x86 core [Shimpi and Wilson 2008]

2.3 GPU Programming Model

Modern GPUs adopt a heterogeneous computing model in which a CPU is in charge of initializing a program and launching parallel computing tasks on one or more GPUs. In this section, we introduce GPU programming model, which is the abstraction for programmers to understand how a program will be executed on a given GPU. This section focuses the latest results, while an earlier review on the general purpose programming models for GPU scan can be found in [McCool 2008].

Despite the architectural differences, a common feature of GPUs is the deployment of a large number of relatively simple processing cores. Compared to the cores in a multi-core CPU, the cores in GPUs have rather lower hardware complexity and much less capacity of cache memory. GPUs actually achieve a high throughput by deploying their cores to handle a large number of data sets concurrently. As a result, all GPUs follow a data-parallel programming model. In this paper, we survey 3 major data-parallel programming models, NVIDIA CUDA, OpenCL, and Intel Ct, but with an emphasis on NVIDIA

CUDA due to its relative maturity. Every programming model is actually correlated to an execution model that specifies how a program is actually executed on computing hardware. To help understand the synergy of hardware and software, we also go through the execution model of CUDA.

2.3.1 NVIDIA CUDA

CUDA is a platform technology, which consists of computing hardware, programming models, and software development toolkits, for data-parallel computing on NVIDIA graphics processors [NVIDIA 2007a]. It is designed to help the development of a GPGPU eco-system. In this sub-section, we review the CUDA programming model.

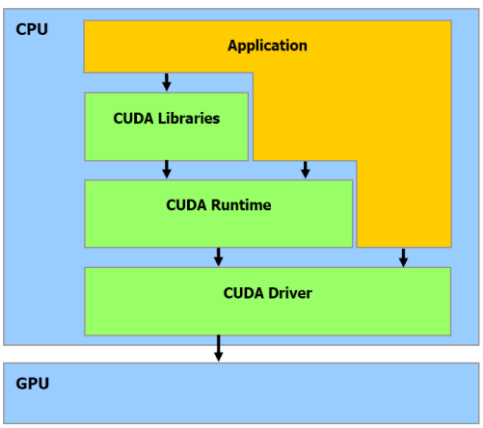


Fig. 2.14 Software stack for CUDA programs [NVIDIA 2007a]

CUDA is an extension of C/C++ language [Stroustrup 1997] by introducing a small set of new keywords, function libraries, and a runtime. A CUDA program is composed of both CPU and GPU code. Coordinated by the sequential CPU code, the GPU code is concurrently executed by batches of threads. A function called by a CPU function but executed on a GPU is called a kernel. One CUDA program may have multiple kernels. A CUDA compiler converts GPU source code into Parallel Thread eXecution (PTX) format [NVIDIA 2007b], which is dynamically re-compiled by the CUDA runtime before execution. The benefit of such an approach is that the same CUDA program can be optimally executed by GPUs with

evolutionary hardware architectures and varying configurations. Besides dynamic compilation, CUDA runtime also offers API functions for initializing GPU devices, launch kernel functions, and data movements between CPU and GPU. All CUDA library APIs and runtime functions will be passed to a CUDA-enabled GPU driver and then finally delivered to GPU hardware. Fig. 2.14 shows the software stack of CUDA.

CUDA adopts a single program, multiple data (SPMD) programming model [Darema 2001]. SPMD is an extension of the well-known single instruction multiple data (SIMD) model. Under the context of SIMD, multiple parallel processes execute the same program, but work on different data. All processes have to go through exactly the same program execution path under the SIMD model, while the SPMD model allows processes to follow various paths. NVIDIA often designates their SPMD approach as single instruction multiple threads (SIMT) because the processing of multiple data is by multiple threads. In a CUDA program, the basic unit of parallel execution is a thread and its internal instructions are sequentially executed on a CUDA core. A CUDA thread is light-weighted because it has direct hardware support (i.e., with dedicated registers and other related logic). As a result, the initialization and switching of a threads incur little overhead. For instance, it takes only one cycle to switch an active thread into the suspended state, and vice versa. On a CPU, however, a similar switching will take hundreds of cycles [Herlihy and Shavit 2008]. A GPGPU application usually launches tens of thousands of threads for massively parallel execution of a program.

A number of threads are organized into thread blocks in a 1-D, 2-D, or 3-D manner. Thread blocks are building blocks for a 1-D, 2-D, or 3-D grid. A kernel always has a corresponding grid. Each thread has a unique index inside a block and each block has a distinctive index in a grid. Such indices are stored as system defined variables. When launching a kernel function, a CUDA programmer must indicate the organization of the corresponding thread block and grid. The arrangement should

be designed to match the problem and/or algorithm structure so as to simplify programming difficulty.

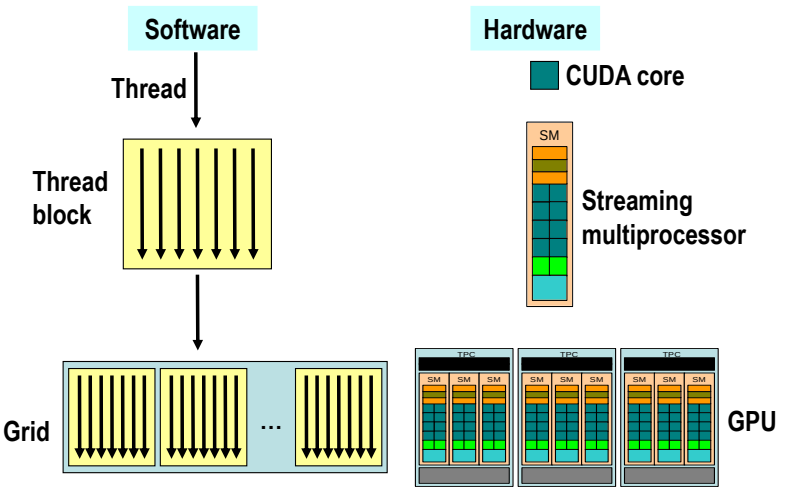


Fig. 2.15 Parallel program organization against hardware hierarchy

The 3-tier thread organization is meant to match the hardware architecture. The corresponding relations are illustrated in Fig. 2.15. Upon activation, a thread starts running on a given CUDA core. During execution, a thread block will be assigned to a certain streaming multiprocessor, while a multiprocessor can accept multiple thread blocks as long as the total number of threads is under a threshold. A global scheduler maps and coordinates a grid of thread blocks to multiprocessor of a GPU.

The threads inside a block are able to collaborate in 2 different ways. First, they can exchange data through the shared memory installed inside a multiprocessor. Second, threads in a block can synchronize with one another by a barrier. If a barrier is set (e.g., via calling a corresponding CUDA runtime function), all threads in a block have to reach the barrier before any of them can execute an instruction after this synchronization point. Threads in different blocks can only communicate through the global memory. Atomic memory operations are supported to maintain data consistency in both the shared memory and the global memory. While the barrier synchronization is from an instruction execution perspective, CUDA also provides a fence synchronization mechanism

focusing on memory operations. The fence can be across a thread block or all threads in a kernel. Given a fence, all the related threads have to wait until the fenced write is finished. Compared with the barrier mechanism, the synchronization power of fence is weaker because it only focuses on memory accesses.

When receiving one or more thread blocks, a streaming multiprocessor of the Fermi GPU groups every 32 (the specific number may vary on different GPUs) threads into a warp. The threads in a warp have the same instruction execution schedule, i.e., a warp of threads always execute the same instruction. On the other hand, modern GPUs support the SPMD processing pattern in which different threads may have varying execution paths. Provided that a CUDA program has conditional code consisting of a pair of "if" and "else", a thread in a warp may take the "if" branch, while another thread follows the corresponding "else" branch. Such a situation is coined as branch divergence, which has a negative impact on the execution efficiency. When it happens, the processing procedures of different branches have to be serialized. Given N divergent branches taken by threads in a warp, if a branch i takes M_i cycles to finish, then the warp finishes after $\sum_{i=1}^{N} M_i$ cycles. Branch divergence is an inherent problem of SIMD machines, but can be mitigated through advanced hardware and/or software techniques. Extensive research efforts have been dedicated to improve the performance of executing SPMD code with branch divergence (e.g., [Fung et al. 2007; Meng et al. 2011; Narasiman et al. 2011; Rhu and Erez 2012]).

To hide the memory latency, a multiprocessor typically maintains a large number of warps on the fly. When any threads in a warp need to access data from the global memory, another warp with their data ready for all threads will resume running in the next cycle. To enable efficient GPU execution, a sufficient number of threads or warps have to be assigned to each multiprocessor. Here an important concept is the multiprocessor occupancy, i.e., the ratio of active warps to the maximally allowed number of warps on a multiprocessor. Note that the number of active threads has to be reduced if a thread

consumes too many registers and/or shared memories. Generally, the overall computing throughput improves with the growing of the occupancy. However, a more complicated thread often offers a higher level of ILP. The tradeoff between thread-level parallelism and instruction level parallelism has rich implications on performance. Volkov [2010] reported cases in which a better level of performance is achieved at a lower occupancy but a higher level of ILP.

A thread block retires when all its threads finish. Then the corresponding multiprocessor can accept another block if there are still unfinished ones. The above mechanism suggests that threads in different blocks actually follow a multiple instruction, multiple data (MIMD) pattern.

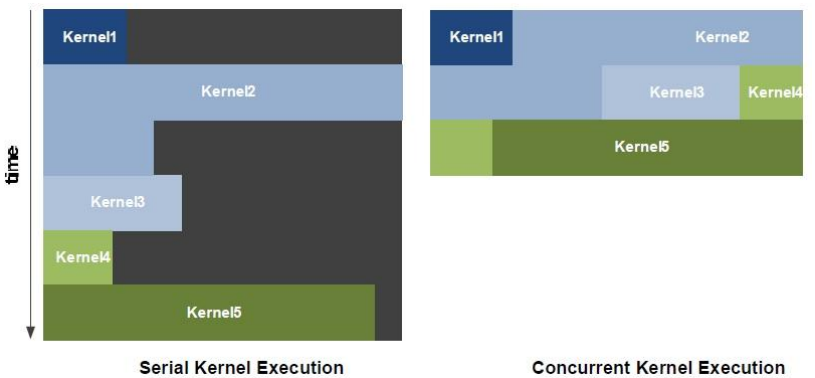


Fig. 2.16 Concurrent kernel execution of GPUs [NVIDIA 2007a]

There are situations that a GPU kernel only needs a small number of multiprocessors and thus cannot fully exploit the computing power of a GPU. In addition, when most blocks of a kernel already end, many multiprocessors on a GPU have to be idle. Accordingly, the Fermi GPU introduces the concept of the concurrent kernel execution. Now several kernels can share the resources of a GPU so that the overall throughput can be sustained. Fig. 2.16 is an illustration of both serial kernel execution and concurrent kernel execution. Fermi GPU uses a fixed preemptive strategy to map kernels to computing resources. If the first kernel cannot occupy all multiprocessors, the second kernel is launched on the remaining cores. Otherwise, if a kernel has workload that consumes all the

resources, no other kernels can be started until there are idle multiprocessors. The resource allocation among concurrent kernels cannot be changed after kernel initialization.

Before a GPU starts computing, the required data have to be transferred from CPU's main memory to GPU's memory. When the GPU finishes computing, the output results need to be copied back to CPU's main memory. The data transfer is through a 16-lane PCI express (PCIe) bus [PCI-SIG 2002], which has an 8GB/s peak bandwidth (4GB/s in each direction). The effective bandwidth, however, is determined by many other factors such as system traffic and data volume. CUDA runtime provides a set of the functions for such transfers. In addition, some applications need to exchange data during the GPU execution. NVIDIA's Fermi GPU supports a zero-copy mechanism so that programmers can directly exchange small volume data with CPU inside a GPU kernels function [NVIDIA 2009]. For large data sets, programmers must re-design their algorithms to include global synchronization points for data transfer.

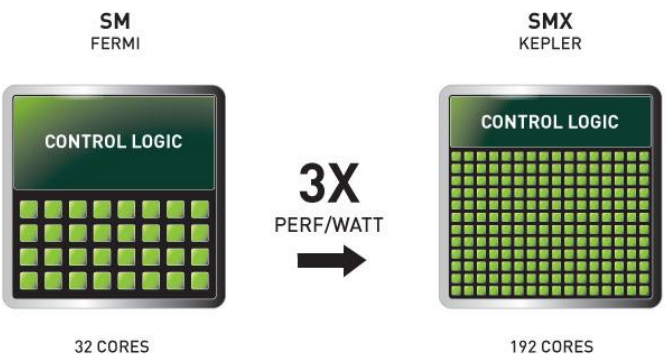


Fig. 2.17 Streaming channels of Fermi and Kepler GPUs [NVIDIA 2012a]

It is time-consuming to move large data sets between CPU and GPU. NVIDIA GPUs are enhanced with a streaming mechanism, so-called asynchronous memory copy, so that the data transfer can proceed in parallel with GPU computing. Under such a paradigm, a thread block can be allocated to a streaming multiprocessor and started working as soon as its

data are already available. This actually follows the stream processing model of computation [Stephens 1995]. The Fermi GPU only allows a single streaming channel, designated as message pass interface (MPI) in NVIDIA's terminology, while the Kepler GPU supports multiple virtual channels for up to 16 kernels to share the physical PCIe bus. A comparison is drawn in Fig. 2.17.

NVIDIA also released a few CUDA based libraries including CUBLAS for dense linear algebra [NVIDIA 2007c], CUFFT for Fast Fourier Transform [NVIDIA 2007d], CUSPARSE for sparse matrix computation [Naumov et al. 2010], CUDPP for data-parallel programming [CUDPP 2010], and so on.

2.3.2 OpenCL

OpenCL is designed to provide an open, royalty-free programming language that is compatible across multiple GPU platforms with varying underlying architectures. The standardization of OpenCL is initialized by Apple and supported by Intel, NVIDIA, AMD, ARM, and other industry participants. The programming model in the current release of OpenCL is specified by the Khronos group [Khronos 2012]. Note that OpenCL is not limited to GPUs and can also be used a programming model for multi-core CPUs.

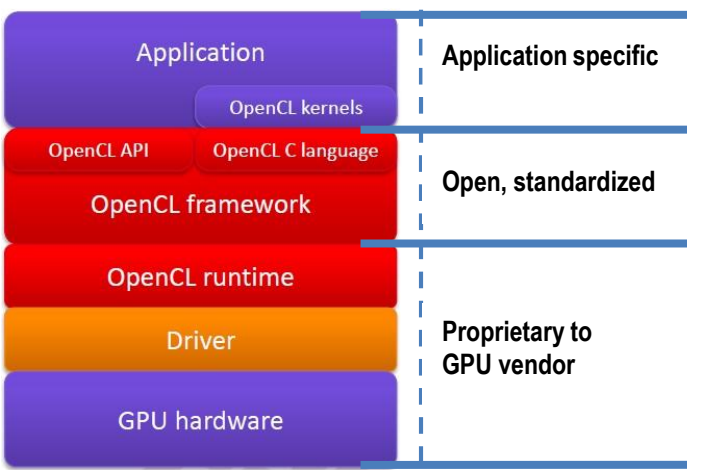


Fig. 2.18 OpenCL software architecture

The software architecture of OpenCL is demonstrated in Fig. 2.18. OpenCL is actually a C-like language plus a set of APIs defined under a unified framework. A runtime handles the API functional call and other service requests. The runtime has to be customized by GPU vendors to ensure function calls from upper levels can be properly passed to their GPU hardware.

Most currently available OpenCL compilers are based on the LLVM compiler infrastructure [Lattner and Adve 2004]. To support compatibility, OpenCL provides a rich set of APIs to locate and identify OpenCL-enabled devices. In addition, a relatively more complex context mechanism is exploited to define the running environment and parameters of kernels.

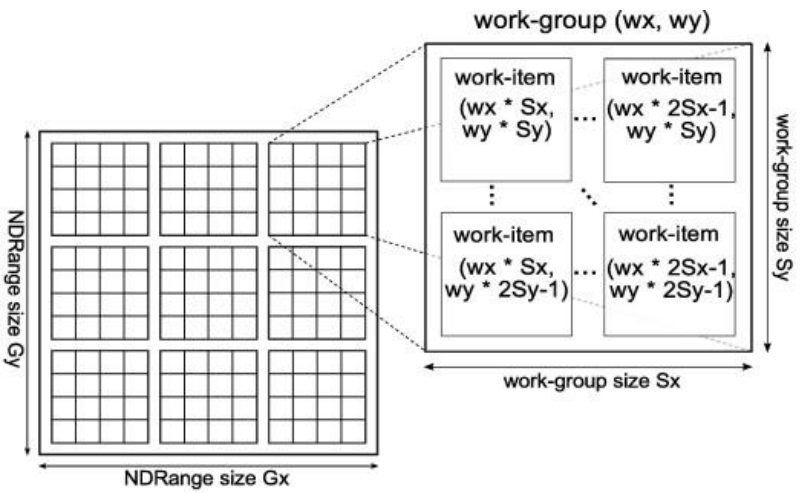


Fig. 2.19 Concurrent workload organization of OpenCL [Khronos 2012]

Similar to CUDA, OpenCL also adopts a 3-tier organization of concurrent work as illustrated in Fig. 2.19. The equivalent concept of CUDA thread is work-item. Multiple work-items constitute a work-group. The organization can be 1-, 2-, or 3-Dimensional. In OpenCL, a kernel function (i.e., the function called on a CPU and executed on a GPU) is designated as a N-Dimensional range (ND-Range or NDRange), which has a 1-, 2-, or 3-Dimensional organization of work-groups.

The memory model of OpenCL is shown in Fig. 2.20. There are 4 different types of memory. Among these, global and

constant memories are common resources to all work-items. They can be cached if the underlying hardware has such capabilities. Local memory is the counterpart of CUDA's shared memory. It is shared by all work-items in a work-group. Private memory is exclusively used by a work-item. It can be implemented as registers, but OpenCL does not enforce hardware implementation of the private memory.

OpenCL can be considered as a generalization of CUDA. The parallel work is organized in a fundamentally equivalent manner in both languages, but OpenCL adopts features to support compatibility.

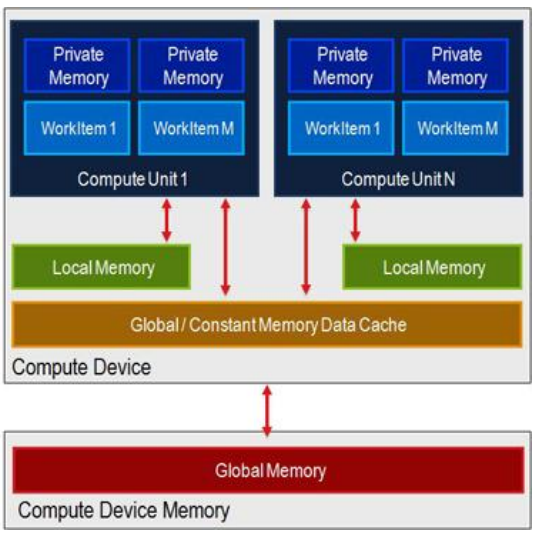


Fig. 2.20 OpenCL memory model [Khronos 2012]

2.3.3 Intel Ct

Intel released a data-parallel programming model, Ct, for many-core GPUs such as Larrabee [Ghuloum et al. 2007]. It is more abstract than CUDA and OpenCL by hiding the underlying hardware threads and vector instructions from the programmers. Accordingly, Ct is architecture independent among Intel CPUs. This means a Ct program runs on Larrabee GPU and other Intel CPUs with SIMD units without re-compilation. Ct was later merged into Intel's Array Building Blocks (ArBB) package [Intel 2010].

Ct is based on the vector computing model [Blelloch 1990], instead of explicitly using a multithreading model. As exemplified in Fig. 2.21, the *TVEC* data type is the basic type of vector in Ct. The memory spaces for data-parallel computing and traditional CPU-based sequential computing are separated. Such a separated memory space enforces the safety of parallel operations on vectors. Before and after data-parallel computing on Ct-enabled hardware, data have to be transferred into and out of vector space. The compiler and runtime transparently map Ct's vector operations to multiple SIMD units located on one or more cores as illustrated in Fig. 2.21. Obviously, the target architecture of Ct is Larrabee alike architectures that consist of multiple CPU cores enhanced with wide SIMD units. Such an automatic approach significantly reduces the programming effort, but leaves little space for programmers to optimize the performance.

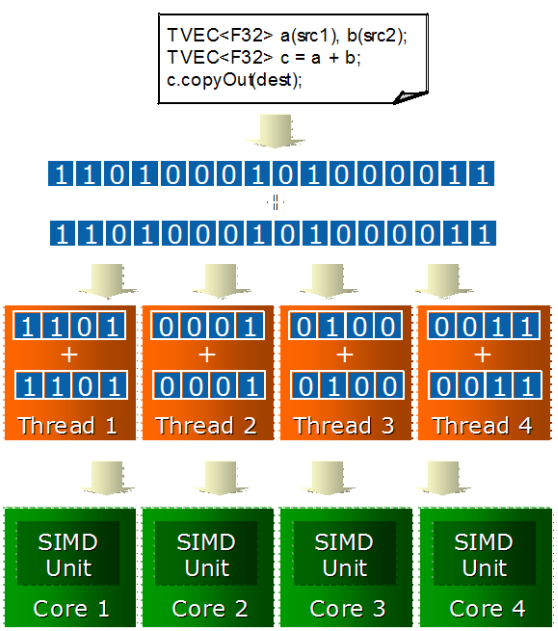


Fig. 2.21. Intel Ct programming model [Ghuloum et al. 2007]

An important feature of Ct is the support of a nested data-parallel model, which is especially amenable to irregular algorithms [Blelloch 1990]. A *TVEC* can have a set of vectors as its elements. Such nested vectors naturally match the

manipulation of irregular data structures such as sparse matrices. **Code** 2.1 is the Ct code for computing sparse matrix vector product. The matrix is stored in a Compressed Sparse Row (CSR) format [Saad 2000].

Code 2.1 Ct code snippet for sparse matrix vector product (adapted from [Ghuloum et al. 2007])

```
TVEC<F64> sparseMatrixVectorProductSC(TVEC<64> Values, TVEC<32>
        RowIdx, TVEC<I32> ColP, TVEC<F64> v)
{
        TVEC<F64> expv = distribute(v, ColP);
        TVEC<F64> product = Value*expv;
        product = product.applyNesting(RowIdx, ctIndex);
        TVEC<F64> result = product.addReduce();
        return result;
}
```

The Ct vectors can only be processed by Ct operators, which can be classified into three categories: element-wise operators, collective communication operators, and permutation operators. An element-wise operator works on two vectors. The operation applies to every pair of elements and there is no interaction between different pairs. Obviously, such operators can be trivially parallelized. A collective communication operator performs computations over all elements in a vector and creates a single value or a new vector. One such example is the *addReduce* operator, which calculates the summation over the whole vector. A permutation operator involves re-arranging the order of elements in a vector.

Due to its relative lack of maturity, Ct and the Larrabee platform have found few applications in EDA problems. Nie [2009] provides an early evaluation on the feasibility of using Ct to implement a few core computing patterns of EDA.

2.3.4 Other GPU Programming Models

There are a few other GPGPU programming models that have been developed. In this sub-section, we give a brief review on these approaches.

An early GPGPU language is Brook+ [Buck et al. 2004]. It adopts a semantics with the flavor of stream processing languages. DirectCompute is released by Microsoft as part of

the DirectX 11 collection of APIs [Microsoft 2009]. It runs on both DirectX 10 and DirectX 11 graphics processing units. It is rather similar to CUDA and OpenCL but defined in a different terminology. Table 2.1 is a comparison of the terminologies defined in these 3 GPGPU languages. The concepts listed in one row are equivalent.

Table 2.1 terminologies defined in these 3 major GPGPU languages

CUDA	OpenCL	DirectCompute	Description
Kernel	Kernel	Compute shader	A program executed by GPU to work on many data items in parallel.
Thread	Work-item	Thread	An instruction stream of the kernel code on a single data item. A thread is usually executed by a SIMD lane.
Thread block	Work-group	Thread group	A group of (potentially) collaborative threads
Grid	N-D range	Dispatch	1-, 2-, or 3-D organization of group of threads

OpenACC is a directive based standard for automatic construction of GPGPU code [OpenACC 2011]. As the equivalent of OpenMP [Chapman et al. 2007] in GPGPU domain, OpenACC is a set of standard pragmas that can be annotated in C/C++ or FORTRAN source code to guide compilers for automatic generation of massively parallel GPGPU code. The standard is maintained by a few industry players including CAPS Enterprise, Cray, The Portland Group and NVIDIA. With the directives, a programmer focuses on exposing parallelism. **Code** 2.2 exemplifies the usage of OpenACC directives in a FORTRAN snippet. Every loop enclosed in a pair of "!$acc kernels" and "!$acc end kernels" will be automatically converted into a kernel function. The compiler assumes the loop can be completely unrolled for parallel execution.

Another directive based approach is OpenHMPP [CAPS 2007], which is actually a superset of the OpenACC. Offered by CAPS Enterprise, OpenHMPP includes such advanced features as the support for multiple GPUs, automatic loop

transformation for higher parallelism, and the integration of hand-crafted CUDA code.

Code 2.2 OpenACC directives used in a FORTRAN code snippet

```
!$acc kernels
do i=1,n
    a(i) = 1.0
    b(i) = 2.0
    c(i) = 2.0
end do

do i=1,n
    a(i) = b(i) + c(i)
end do
!$acc end kernels
```

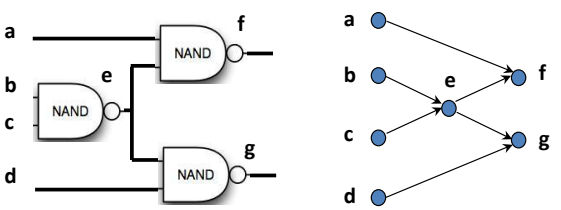
3. EDA COMPUTING PATTERNS

To deal with the ever-rising complexity of IC chips, EDA toolkits have grown into one of the most complex software systems. The modern IC design process involves a wide range of problems originated from diverse domains. As a result, EDA software exploits a large number of algorithms of diversified natures. It is thus infeasible to parallelize EDA software on an application-by-application base. Fortunately, recent research shows that the complicated EDA software is actually based on a few basic computing patterns [Catanzaro et al. 2008]. A pattern based parallelization approach [Mattson et al. 2004] turns out to be essential because complicated software can be constructed as a composition of basic computing patterns. According to a review on 17 major categories of EDA applications, Catanzaro et al. [2008] identified 7 key computing patterns that were pervasive in EDA applications. Below is a description of these patterns.

– *Dense Linear Algebra*. The pattern is widely used by EDA applications such as delay calculation, power grid analysis, and various optimization problems. The basic operations of dense linear algebra are defined as Basic Linear Algebra Subprograms (BLAS) Level 1, 2, and 3 routings [BLAS 2008], i.e., vector-vector, matrix-vector, and matrix-matrix operators. Typical problem instances include linear system solution, linear programming, Newton-Raphson method, non-linear optimization, and so on.

– *Sparse Linear Algebra*. A matrix is sparse when only a small percentage of its elements have a non-zero value. One example is the adjacency matrix for a VLSI netlist pervasively used by many EDA applications. As illustrated in Fig. 3.1, a matrix entry *(i, j)* encodes the interconnection information between cells *i* and *j*. The entry *(i, j)* has a non-zero value if and only if when the output of cell *i* is fed to cell *j*. Obviously, a cell usually has only a few connected cells in a whole netlist and thus the matrix tends to be very sparse. Such sparsity can be exploited to reduce the memory requirement and/or enhance computing efficiency by using compressed storage formats [Saad 2000].

However, sparse matrix based algorithms and data structures are generally more complex than their dense matrix equivalents. In addition, sparse matrices are usually correlated with strong irregularity, which means that the memory access patterns and computation intensity cannot be statically determined. For instance, the number of non-zeros in different rows may vary substantially and it is thus difficult to maintain load balance in a static fashion. Typical problems fallen into this category include sparse-matrix vector product, linear and nonlinear system solution, differential equations, etc.



	a	b	c	d	e	f	g
a	0	0	0	0	0	1	0
b	0	0	0	0	1	0	0
c	0	0	0	0	1	0	0
d	0	0	0	0	0	0	1
e	0	0	0	0	0	1	1
f	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0

(c)

(a) A simple logic circuit, (b) Directed graph corresponding to the circuit (a), (c) Adjacency matrix corresponding to (b)
Fig. 3.1. Sparse matrix

– *Graph Algorithms*. Graph algorithms are very popular in EDA applications. A VLSI circuit is naturally represented as a graph to be manipulated by EDA algorithms. In addition, many EDA problems can be formulated as various graphs. Graph traversal is one major category of graph algorithms. Problems such as breadth-first search, depth-first search, shortest path, and all-pair shortest path

belong to this category. The levelized, zero-delay logic simulation technique also follows a graph traversal pattern [Lewis 1991]. Another category of graph problems modify the input graph. One example is the construction of a minimum-spanning tree, which is used in many routing algorithms [Cormen et al. 2001]. Another example is the graph re-writing algorithm used in high level synthesis [Huijs 1996].

- *Backtrack and branch-and-bound*. Both backtrack and branch-and-bound are general search algorithms to find optimal solutions to combinatorial problems [Cormen et al. 2001]. They constitute the backbone of ATPG, BDD and SAT algorithms. The backtrack approach incrementally refines a partial solution by keeping track of the assignment of variables in a decision tree. If a partial solution turns out to be infeasible, the algorithm abandons it and tries another one. Branch-and-bound takes a similar approach, but filter out infeasible solutions through quickly evaluating the upper and lower bounds of a solution. It should be noted that these two algorithms have an exponential complexity in the worst case.
- *MapReduce*. Unlike other EDA patterns, MapReduce is inherently a parallel design pattern [Dean and Ghemawat 2004]. Originally designed for large data analytics application, it is actually a distributed computing model taking a divide-and-conquer strategy. The "map" step decomposes input data and assigns each chunk to different computing devices. Then the "reduce" step merges the partial results to come up with a complete solution. The trial and evaluation process of simulated annealing can be seen as an instance of the MapReduce pattern.
- *Dynamic programming*. Dynamic programming is also a generic optimization algorithm that takes a divide-and-conquer approach [Cormen et al. 2001]. To successfully apply this algorithm, a target problem must have an "optimal substructure", i.e., the optimal solution is the combination of optimal solutions to all sub-problems. Dynamic programming is a well-known technique for the

technology mapping problem. When a circuit and all gates in the technology library can be decomposed into trees of NAND2-INV circuits, the technology mapping problem can be optimally solved by a dynamic programming technique [Keutzer 1987].

– *Structured grid*. In this pattern, identical operations are performed on a regularly organized grid. Detail routing problems fall into this category. Many routing heuristics are applied to each grid to complete the wiring of every net inside. This pattern can be generalized to the case of handling a multi-dimensional array of objects. One typical application is the assembling of matrix elements for circuit simulation [Gulati et al. 2009].

The 7 patterns cover almost all EDA applications as summarized in Table 3.1. It can be seen that graph algorithms, sparse and dense linear algebra, and backtrack and branch-and-bound are the most frequently used patterns of EDA computing. Perhaps the only important EDA application that cannot be fully covered is logic simulation of digital circuits. Although the levelized simulation algorithm [Lewis 1991] for cycle-accurate, zero-delay simulation falls into the graph algorithm pattern, the more general timed simulation problem is completely dynamic and does not follow an explicit pattern [Pingali et al. 2011].

We also analyzed the underlying data structures of the 7 design patterns. Among them, both dense linear algebra and structured grid patterns possess a regular data structure. Basically, the memory access patterns are uniform across all data objects. Such data structures are very friendly to GPU platforms for two reasons. First, the regular distribution of requested memory addresses can be easily coalesced to hide memory latency. Second, the fully predictable memory access pattern allows programmers to partition the work for a good load balance. For example, in the matrix-matrix multiplication problem, it is very natural to assign one thread to calculate one element in the product matrix. The resultant organization of the parallel work is both intuitive and efficient.

However, the remaining 5 design patterns are generally based on irregular data structures. The EDA problems to be solved by these patterns are inherently based on irregular data structures. The most typical irregular data structures in EDA include both sparse matrices and graphs. Out of the 17 major EDA applications investigated in [Catanzaro et al. 2008], 15 applications are built on top of graph algorithms and 4 applications involved sparse matrix computations. In addition, the 17 applications do not include device physics and process simulations, which also involve large scale sparse matrices for finite element computations.

Table 3.1 Essential design patterns for EDA computing (based on the work by Catanzaro et al. [2008] and our analysis)

Computing Pattern	**Related EDA Applications**	**Regularity of Data Structures**
Dense linear algebra	Delay modeling, placement, circuit simulation, power grid analysis, circuit optimization	Regular
Sparse linear algebra	Delay modeling, timing analysis, placement, circuit simulation	Generally irregular
Graph algorithms	HDL synthesis, technology independent optimization, technology mapping, timing analysis, sequential circuit optimization, floorplanning, placement, routing, DFM, design rule checking and compaction, model checking, equivalence checking, delay testing, levelized logic simulation	Generally irregular
Backtrack and branch-and-bound	Technology independent optimization, timing analysis, routing, model checking, equivalence checking, ATPG	Irregular
MapReduce	Floorplanning, placement, DFM	Generally irregular
Dynamic programming	Routing, technology mapping	Generally irregular
Structured grid	Placement, power grid analysis	Regular

The irregular data access patterns are determined by the very nature of VLSI circuits. In a typical gate level netlist, while

most gates only connect to a small but non-fixed number of neighboring cells, certain gates may have hundreds of fan-outs. Hence, the resultant data structures to encode the netlist have to be irregular. One example is the connection matrix required by the quadratic placement and force-driven placement (e.g., [Kleinhans et al. 1991; Eisenmann and Johannes 1998]). In such matrices, an element (*i*, *j*) is non-zero if and only if cells *i* and *j* are connected by a wire. According to our experiments on ISPD2006 benchmark circuits [Nam et al. 2007], the connection matrices are extremely sparse. Most rows of such matrices only have 3 to 5 non-zeros among millions of entries. However, there do exist a very small number of rows that have hundreds or thousands of non-zeros. Such rows correspond to clock and other special signals

Although it has long been known that sparse matrix operation and graph algorithms possess sufficient data level parallelism [Blelloch 1990], it is extremely challenging to find an efficient implementation of them on GPUs for two reasons. First, most die area of a GPU is dedicated to computing resources and only a very small capacity of cache is available. For irregular applications where the memory access patterns are unpredictable, GPUs may have difficulty to hide memory latency. Second, it is hard to maintain load balance among processing cores of a GPU when processing a non-uniform distribution of data. The load balance can be exhibited on 2 different levels. One concern is that the imbalanced workload inside the warp leads to inefficient usage of SIMD lanes. Another concern is that a thread block may have to wait for a small number of slow warps. Then the corresponding hardware resources on the multiprocessor cannot be released.

4. ACCELERATING KEY DESIGN PATTERNS ON GPUS

In the last chapter, 7 fundamental computing patterns of EDA applications were identified. We review GPU based parallel computing techniques for these patterns in this chapter. Complex EDA applications can be constructed by composing these computing patterns.

4.1 Dense Linear Algebra

Dense linear algebra involves operations on dense matrix, which is a comparative concept against sparse matrix. Here it means regular matrix manipulated in a straightforward 2-diemensional fashion. Note that we often use a 1-D array to store a matrix by sequentially putting each row or column into the array. Dense matrices are ideal for GPGPU implementations. In fact, many graphics applications are inherently built on top of this design pattern. When performing computations on a dense matrix, the memory access pattern is usually regular and known a priori. Therefore, the memory latency can thus be effectively hidden through memory coalescing. Meanwhile, the computing process can generally be organized as a 2-D grid of threads with each thread operating on a certain part of a matrix. Accordingly, a good load balance is trivially achieved. In many science and engineering domains, dense linear algebra applications already enjoyed considerable speedup when ported to GPUs. Interested readers please refer to [Owens et al. 2005; Owens et al. 2008] for excellent reviews on early works in this direction. In the remaining of this section, we survey recent progress in implementing EDA related dense linear algebra on GPUs. We mainly focus on 2 fundamental problems, matrix-matrix multiplication and linear system solution.

4.1.1 General Dense Linear Algebra Operations

General dense linear operations include 3 levels of BLAS procedures, i.e., vector-vector, matrix-vector, and matrix-matrix procedures [BLAS 2008]. They play a fundamental role in the science and engineering computing applications.

Therefore, a lot of works have been dedicated to developing high performance libraries. The most notable works include CUBLAS [NVIDIA 2012], MAGMA [Nath et al. 2010] and CULA [Humphrey et al. 2010]. These libraries are highly tuned for a series of GPU architectures. They are also designed with well-defined function interfaces to be reused in complex applications. We focus on CUBLAS as an example to introduce core techniques.

Code 4.1 Code skeleton based on CUBLAS

```
1  int main( void ){
2          float* host_vector, * device_vector;
3
4          host_vector = (float*) malloc(M*sizeof(float));
5
6          ... // Initialize vector of M floats
7          cublasAlloc(M, sizeof(float), (void**) &device_vector);
8
9          cublasSetVector(M, sizeof(float), host_vector, device_vector, 1);
10         cublasSscal(M, ALPHA, device_vector, 1);
11         cublasGetVector(M, sizeof(float), device_vector, host_vector, 1);
12
13         cublasFree(device_vector);
14         ….
15}
```

CUBLAS [NVIDIA 2012] is built on top of CUDA and NVIDIA GPU driver APIs with the similar function interfaces as BLAS [BLAS 2005]. It adopts a vector based programming style. **Code** 4.1 illustrates how to use CUBLAS to scale a vector. The program calls CUBLAS functions to create and destroy vector and matrix objects, fill them with data, and conduct numerical operations. Lines 7 and 13 allocate and free space for the vector in the GPU memory. Lines 9 and line 10 transfer the data between CPU and GPU memory. The function call in line 10 conducts the scaling operation for the given vector. When calling the function, the corresponding operation is performed over all vector elements.

With the introduction of each generation GPU hardware, the library was upgraded to exploit new hardware features for better performance. The latest released CUBLAS5.0 targets the Fermi architecture. The so-called CGEMM (defined by [BLAS 2008]) kernel can sustain a throughput level of up to 1TFLOPs,

about 75% of the peak single precision performance of Tesla M2090 accelerator card. For double precision data, the ZGEMM kernel achieves a throughput of 400GFLOPs. The simple function interfaces and high computing throughput substantially lower the development difficulty of GPU numerical applications. For instance, Yu et al. [2009] reported that it only took a few days to port the device model of a nano-tube to a NVIDIA GPU and achieve a 7X speed-up.

The matrix-matrix operators, especially general matrix multiplication (GEMM), are usually the performance bottleneck of a CUBLAS based program. Many efforts have been made to improve the GEMM performance ([Volkov 2008; Kurzak et al. 2010; Tan et al. 2011; Li et al. 2012]). Therefore, we take GEMM to exemplify the optimizing and tuning techniques adopted by CUBLAS.

Code 4.2 C code for computing matrix multiplication

```
void gemm_serial(int *A, int *B, int *C, int m, int k, int n)
{
  for (int a = 0; a< m; a++)
      for(int b = 0; b < n; b++){
          C[a*n+b] = 0;
          for (int c = 0; c < k; c++)
                C[a*n+b] += A[a*k+c] * B[c*n+b]
      }
  }
}
```

The serial C code to calculate $C=AB$, where A, B and C are matrices with dimension as $m \times k$, $k \times n$ and $m \times n$, respectively, is listed in **Code** 4.2. The serial code consists of three nested loops. When implementing it on GPU, we can unroll the two outer loops. Each thread calculates an element of matrix C, i.e. the inner loop. A block computes a row of matrix C. As all threads run concurrently, a large number of elements of matrix C can be calculated simultaneously. The GPU code is illustrated in **Code** 4.3.

Code 4.3 Naïve GPU implementation for GEMM

```
__gloabl__void gemm_navie(int *A, int *B, int *C,int m, int k, int n)
{
  int row = blockIdx.x;
  int col = threadIdx.x;
  int pos = row*n + col;
  int Elem = 0;
  for(int i = 0; i < k; i++)
      elem += A[row*k + i] * B[i*n+col]
  C[pos] = elem;
}
```

Code 4.3 exploits the inherent thread level parallelism. However, it incurs a huge number of requests to GPU memory. Every element in matrix *A* is accessed for *n* times, while an element in matrix *B* is read for *m* times. Since the matrices are stored in GPU global memory, such a huge number of memory references pose significant performance overhead. This problem can be substantially mitigated by a so-called block matrix multiply technique, developed by Volkov and Demmel [2008]. Here matrices are decomposed into small tiles and each tile is processed by a thread block. By properly choosing the tile size, all tile elements can be pre-fetched into the shared memory so that all threads in a block can share the data. The source code is listed in **Code** 4.4.

In the tile-based implementation, each block is responsible for computing one tile of matrix *C*. The tile is defined as a square with the dimension as BLOCK_SIZE (i.e. 16) and each thread in the block is responsible for computing one element of each tile. Lines 26 and 27 load the data from global memory to shared memory. Each thread is responsible for loading one element of each tile. Line 32 executes the computing operations and accumulates the product of two tiles into a register. Line 37 writes the data in registers into global memory. By pre-fetching, we can take advantage of fast shared memory and reduce the number of global memory accesses. Now we only need to load matrices *A* and *B* are read from global memory for *n/BLOCK_SIZE* and *m/BLOCK_SIZE* times, which is considerably smaller than the naïve GPU implementation does.

Code 4.4 GPU code for tile-based GEMM [Volkov 2008]

```
1 __gloabl__ void tiled_gemm (int *A, int *B, int *C,int m, int k, int n)
2 {
3   //Block Index
4   int blockRow = blockIdx.y;
5   int blockCol = blockIdx.x;
6   //Thread Index
7   int threadRow = threadIdx.y;
8   int threadCol = threadIdx.x;
9
10 //Index of the first and last tile of A processed by the block
11 int aBegin = BLOCK_SIZE * blockRow * k;
12 int aEnd = aBegin + k – 1;
13 //Index of the first and last tile of B processed by the block
14 int bBegin = BLOCK_SIZE * blockCol;
15
16 //Step size used to iterate
17 int aStep = BLOCK_SIZE;
18 int bStep = BLOCK_SIZE * n;
19
20 float Csub = 0;
21 //Loop over all the tiles of A and B to calculate the tile of C
22 for(a = aBegin, b = bBegin; a <aEnd; a += aStep, b += bStep){
23         __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];
24         __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];
25         //load the each tiles of A and B to the shared memory
26         As[threadRow][threadCol] = A[a + threadRow * k +
threadCol];
27         Bs[threadRow][threadCol] = B[b + threadRow * n +
threadCol];
28         __synchreads();
29
30         //Multiple the two tiles
31         for (int c = 0; c < BLOCK_SIZE; c++)
32                 Csub += As[threadRow][c] * Bs[c][threadCol];
33         __syncthreads();
34         }
35 //Write the tile of C to global memory
36 int cBase=BLOCK_SIZE*blockRow* n+BLOCK_SIZE * blockCol;
37 C[cBase + BLOCK_SIZE * threadRow + threadCol] = Csub;
38 }
```

In the tile-based implementation, frequent synchronization operations are required because the shared memory has to be filled before the contents can be used. Each synchronization tends to stall the execution of some threads and lower the

Code 4.5 GPU code for optimized GEMM [Volkov 2010]

```
1 __gloabl__void optimized_tiled_gemm (int *A, int *B, int *C,int m,
int k, int n)
2 {
3  //Block Index
4  int blockRow = blockIdx.y;
5  int blockCol = blockIdx.x;
6  //Thread Index
7  int threadRow = threadIdx.y;
8  int threadCol = threadIdx.x;
9
10 //Index of the first and last tile of A processed by the block
11 int aBegin = BLOCK_SIZE * blockRow * k;
12 int aEnd = aBegin + k – 1;
13 //Index of the first and last tile of B processed by the block
14 int bBegin = BLOCK_SIZE * blockCol;
15
16 //Step size used to iterate
17 int aStep = BLOCK_SIZE;
18 int bStep = BLOCK_SIZE * n;
19
20 float Csub[2] = {0, 0};
21 //Loop over all the tiles of A and B to calculate the tile of C
22 for(a = aBegin, b = bBegin; a <aEnd; a += aStep, b += bStep){
23 __shared__ int As[BLOCK_SIZE][BLOCK_SIZE + 1];
24 __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE + 1];
25 //load the each tiles of A and B to the shared memory
26 As[threadRow][threadCol] = A[a + threadRow * k + threadCol];
27 Bs[threadRow][threadCol] = B[b + threadRow * n + threadCol];
28        As[threadRow+8][threadCol]=A[a+(threadRow+8)*k+
threadCol];
29        Bs[threadRow+8][threadCol]=B[b+(threadRow+8)*n+
threadCol];
30
31        __synchreads();
32
33        //Multiple the two tiles
34        for (int c = 0; c < BLOCK_SIZE; c++)
35                Csub[0] += As[threadRow][c] * Bs[c][threadCol];
36                Csub[1] += As[threadRow+8][c] * Bs[c][threadCol];
37                __syncthreads();
38        }
39 //Write the tile of C to global memory
40 int cBase=BLOCK_SIZE*blockRow* n+BLOCK_SIZE * blockCol;
41 C[cBase + BLOCK_SIZE * threadRow + threadCol] = Csub[0];
42 C[cBase + BLOCK_SIZE * (threadRow+8) + threadCol] = Csub[1];
43 }
```

effective parallelism. Volkov [2010] proposed a technique to extract a higher level of effective parallelism by reducing the block size and increasing the number of blocks. A smaller size of blocks suggests that less threads are to be stalled by a synchronization. Meanwhile, each thread has more work to do so that the two warp schedulers in a multiprocessor of Fermi GPU have a better chance to extract more instruction level parallelism. **Code** 4.5 lists the source code for the optimized implementation. The code highlighted with red fonts indicates the enhancements on the basis of **Code** 4.4. In this optimized implementation, the size of each block is reduced from 16×16 to 16x8. Therefore, each thread has to calculate two elements of each tile of matrix *C*, which is executed in lines 35 and 36. As the operands of these two instructions are independent, they can execute concurrently. An extra benefit of this technique is the possibility of reusing data in the shared memory. In lines 35 and 36, the code accesses the same data in the shared memory array designated as *Bs*. Experimental results show that the overall computing throughput can be increased to 27% by such an optimization [Volkov 2010]. Here, we only show the scenario of computing two elements by every thread. In fact, it can be more beneficial to compute even more elements by one thread. Another optimization technique is to pad the arrays of shared memory allocated in lines 23 and 24 with one extra column.

As on-chip registers are much faster than the shared memory, Tan et al. [2011] proposed a two-level blocking technique to alleviate the pressure for shared memory bandwidth. The abovementioned blocking technique by Volkov and Demmel [2008] reduces the memory traffic to global memory. Tan et al. added the second level, so-called register blocking, to take advantage of registers. According to the number of registers in each SM, an optimal configuration of register block was found to be 4×4. A double-buffering strategy was also developed to overlap the computation and memory operations. With this strategy, a data block (i.e., a tile) is spitted into two halves. When the first half is being used for computation, the instructions to fetch the second half are issued. Since they operate on different buffers in the shared memory, memory

operations can proceed with computation at the same time. These techniques improve the GEMM performance by 20% on the basis of CUBLAS 4.0. For double-precision matrix-matrix multiplication, the achieved peak performance is up to 362 GFLOPs on a Tesla C2050 GPU.

4.1.2 Linear system solution

The numerical solution of linear equations, $Ax=b$, where A is a matrix, and x and b are column vectors, is the core operation of many EDA applications. In this sub-section, we focus on solving dense linear systems, while the respective problem on sparse linear systems will be covered in Section 4.2. Typical EDA applications falling into this category include device simulation [Seoane and Garcia-Loreiro 2005] and power grid analysis [Feng and Li 2008; Feng and Li 2010].

A linear system can be solved by either direct methods or iterative methods. The former use factorization techniques such as LU and QR [Saad 2000] to decompose the matrix into two triangular ones, which can be with significantly less effort. If a matrix is symmetric positive definite, then a Cholesky factorization [Saad 2000] can be used to proficiently find a solution. An iterative method gradually improve an approximated solution through a series of vector and matrix operations. The bottleneck of iterative solvers is the matrix-vector multiplication, which can be readily addressed by the techniques introduced in the previous sub-section. The multigrid method [Briggs 1987] is orthogonal technique that iteratively works on multiple scales of a problem.

4.1.2.1 Factorization methods

Early attempts of implementing LU, QR, and Cholesky factorization methods on GPU hardware were reported in [Galoppo et al. 2005; Jung and O'Leary 2006]. With the evolution of GPU hardware, a variety of techniques have been proposed for different variants of these factorization methods. (e.g., [Agullo et al. 2011; Baboulin et al. 2012; Barrachina et al. 2009; Kim et al. 2012; Volkov and Demmel 2008]). As the optimization techniques for LU, QR and Cholesky are very

similar, we will review the GPU based implementations for the fundamental building blocks of these methods.

Algorithm 4.1Vectorized Cholesky factorization (adapted from [[Jung and O'Leary 2006])

for k=1 to n-1 do

$A_{k,k} = \sqrt{A_{k,k}}$

$A_{k+1:n,k} = A_{k+1:n,k} \big/ A_{k,k}$

for j=k+1 to n do

$A_{j,k+1} = A_{j,k+1} - A_{j,1:k}^{T} \bullet A_{k+1,1:k}$

end for

end for

$A_{n,n} = \sqrt{A_{n,n}}$

First, let us exemplify the factorization procedure with the Cholesky decomposition shown in **Algorithm** 4.1. Given a symmetric positive definite matrix, A, its Cholesky factorization is defined as $A=LL^T$, where L is designated as the Cholesky factor of A. After L is derived, the original linear system, $Ax=b$, can be solved in a straightforward manner. As illustrated in Fig. 4.1, the Cholesky factorization algorithm can be formulated with the vectorized notation as shown in **Algorithm** 4.1, where $A_{k+1:n,k}$ is the column vector containing the $(k+1)th$ to nth elements in the kth column of matrix A, and $A_{j,1:k}$ is the row vector composed of the first k elements of the jth row in matrix A. The procedure is executed in-place, in the sense that the Cholesky factor L, will be stored in the lower triangular of A after completion. The algorithm involves 4 major operations, square root, vectorized division, inner product, and matrix subtraction. With CUBLAS, the above algorithm can be easily implemented.

Similar to Cholesky factorization, the LU factorization of a matrix A computes matrices L and U such that $A=LU$, where L is a low triangular matrix and U is an upper triangular. When the factorization is completed, the original linear system A$x=b$ can be expressed by $LUx=b$, which is easily calculated by solving two triangular systems, $Ly=b$ and $Ux=y$. When the matrix is symmetric positive, LU factorization is reduced to Cholesky factorization. Three classical variants of LU

factorization are left looking, right looking and Crout methods [Dongarra et al. 2001]. We mainly consider the right-looking algorithm as it exposes more thread-level parallelism.

QR factorization decomposes a matrix *A* into the product of a unitary matrix *Q* and an upper triangular matrix *R*, i.e., *A*=*QR*. Several one-sided factorization methods are proposed to compute the QR decomposition, such as modified Gram-Schmidt, Fast Givens and Householder [Kerr et al. 2009]. In this sub-section, we emphasize the Householder algorithm which exhibits a high degree of parallelism.

Similar to CUBLAS, a few GPU based libraries have also been proposed to provide well encapsulated BLAS operations. MAGMA [Nath et al. 2010] is a dense linear algebra library similar to LAPACK [Anderson et al. 1990], but implemented for heterogeneous architectures. The current MAGMA distribution focuses on multicore systems associated with a single GPU card. The main idea is to off-load the compute-intensive operations found in LAPACK algorithms (e.g., BLAS level 3 operations) to the GPU. CULA [Humphrey et al. 2010] is also a high performance linear algebra library that executes in a unified CPU/GPU hybrid environment.

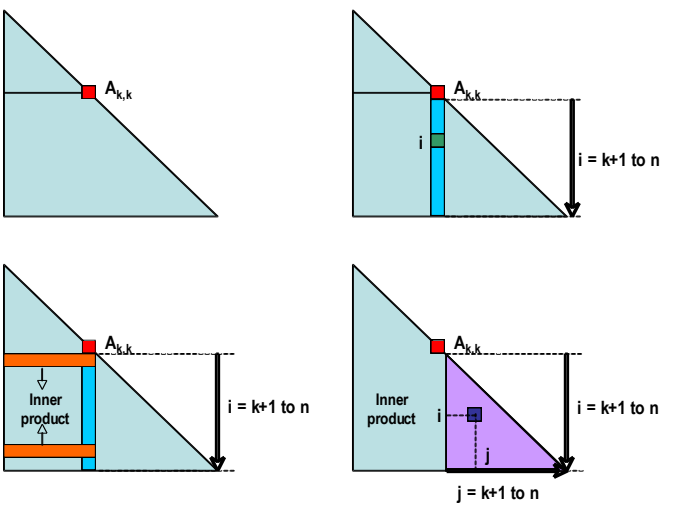


Fig. 4.1 Vectorized Cholesky Factorization (adapted from [Jung and O'Leary 2006]

A straightforward implementation for Cholesky factorization is

depicted in Fig. 4.1. However, the performance of this naïve method is extremely poor on GPU platforms. A similar situation also happens in the case of LU and QR factorizations. To effectively leverage GPU's computing power for factorization methods, a variety of advanced optimization strategies are essential. Such techniques can be classified into the following three directions.

- **Tile-based Processing**

The tile-based technique is often effective to extract a higher level of parallelism in the factorization process. The core idea is to divide an $n \times n$ matrix A into a number of small $b \times b$ sub-matrices (i.e. tiles) such that A consists of $n_b \times n_b$ tiles, where $n_b = n/b$. The matrix A can be expressed as a combination of sub-matrices $A_{i,j}$ $(i <= n_b-1, j <= n_b-1)$.

$$A = \begin{pmatrix} A_{(0,0)} & A_{(0,1)} & \cdots & A_{(0,n_b-1)} \\ A_{(1,0)} & A_{(1,1)} & \cdots & A_{(1,n_b-1)} \\ & \cdots\cdots & & \\ A_{(n_b-1,0)} & A_{(n_b-1,1)} & \cdots & A_{(n_b-1,n_b-1)} \end{pmatrix}$$

The tile-based algorithm was originally designed to improve cache efficiency. A tile with proper size can stay in cache without being flushed and thus there is a better chance of cache reuse. Such an idea obvious also applies to the shared memory of GPUs. In addition, a tiled algorithm naturally decompose a computing job into smaller tasks that can be concurrently processed. Such features make the tile-based technique suitable for many-core GPUs.

Kurzak et al. [2010] implemented a straightforward tile-based QR factorization on GPU. It works in an iterative manner. At the first iteration, the algorithm computes a QR factorization for $A_{0,0}$. The resultant $A_{0,0}$ is then used to update the set of tiles on to the right side of $A_{0,0}$ in an embarrassingly parallel way. As soon as a tile-update in the ith row is completed, its neighbor in the $(i+1)$th row can be started. One can imagine the execution as falling columns of dominos from the top of matrix.

Kim et al. [2012] proposed an efficient tile-based LU factorization method without the need for control-intensive pivoting operations. The algorithm consists of three kernels, TRSM, GEMM and direct LU, where TRSM and GEMM are standard BLAS routines and direct LU is a straightforward implementation of LU factorization. An input matrix A is partitioned into small tiles, which are executed by different kernels independently during each iteration. Fig. 4.2 shows the processing flow of three iterations. At the first iteration, the following computations are conducted.

- $A_{0,0}$ is factored with a straightforward factorization method.
- Tiles $A_{i,0}$, $i=1,\ldots,n_b-1$ are overwritten by $A_{i,0}U_{00}^{-1}$. It is implemented as a solution of triangular linear systems with multiple right-hand sides and can be done by BLAS routine TRSM.
- Tiles $A_{0,k}$, $k=1,\ldots,n_b-1$ are overwritten by $L_{00}^{-1}A_{0,k}$. It is also a triangular solution with multiple right-hand sides and matches the TRSM function.
- Tiles $A_{i,k}$, i, $k=1,\ldots,n_b-1$ are overwritten by $A_{i,k} - A_{i,0}A_{0,k}$. This operation requires a matrix-matrix multiplication that can be done by BLAS routine GEMM.

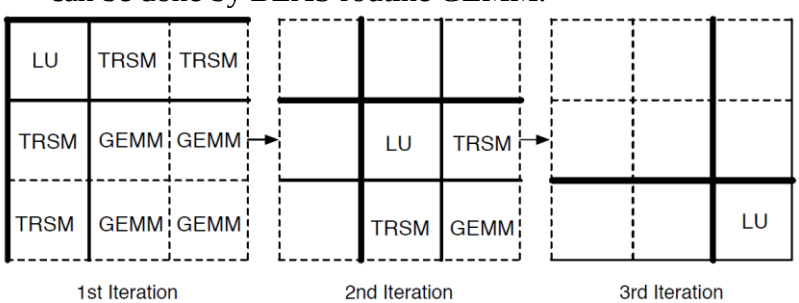


Fig. 4.2 Concurrent tasks during Tile LU factorization (adapted from [Kim et al. 2012]

A better way to describe the iteration operations illustrated in Fig. 4.2 is using the Formal Linear Algebra Methods Environment (FLAME) notation, which is an advanced notation system for matrix computation by hiding details of subscripts. Interesting readers please refer to related documents [Gunnels et al. 2001]). The parallelism is guaranteed by the fact that different tiles can be processed

independently in each iteration. The TRSM and GEMM computation are directly executed on GPU through calling CUBLAS APIs.

Song et al. [2012] introduced a tiled factorization algorithm for heterogeneous platforms. A salient feature of this work is that both CPU and GPU are exploited for computing. It was proved that a uniform tile size did not work well for both CPU and GPUs. A large tile may be too overwhelming for a single CPU core, but a small tile leads to insufficient use of a GPU multiprocessor. The tiled algorithm was extended by considering two types of tiles, smaller tiles for GPU multiprocessors, and larger tiles for CPU. Fig. 4.3 depicts two matrices decomposed into a set of small and large tiles. The scheduling is of special importance to such a heterogeneous computing model. After a tile finishes processing, a few neighboring tiles become ready for computation. In order to evenly allocate the workload between CPU and GPU, a two-level 1-D block cyclic distribution strategy was proposed. The basic idea is to first map a whole matrix to GPU using a 1-D column block cyclic distribution, and then cut a slice of an appropriate size from each block and assign it to the host CPU. The task mapping strategy is applied to both Cholesky and QR factorization as illustrated in Fig. 4.4 and Fig. 4.5, respectively. The algorithm exhibits excellent scalability. A nearly constant GFLOPs-per-core and GFLOPs-per-GPU values can be attained on hardware configurations consists of one to nine CPU cores and three GPUs.

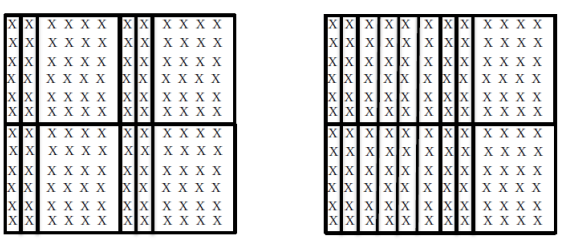


Fig. 4.3 Matrices consisting of a mix of small and large tiles. (a) A 12x12 matrix is divided into eight small tiles and four large tiles. (b) A 12x12 matrix is divided into sixteen small tiles and two large tiles (adapted from [Song et al. 2012])

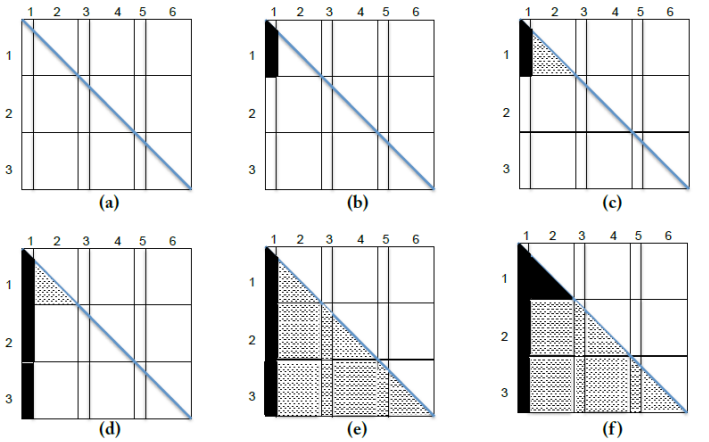


Fig. 4.4 Operations of heterogeneous tile Cholesky factorization. (a) The symmetric positive definite matrix *A*. (b) Factorize the diagonal tile to solve L_{11}. (c) Apply L_{11} to update its right A_{12} by GEMM. (d) Compute TRSMs for the two tiles below L_{11}. (e) Apply GEMM to update all tiles on the right of the first tile column. (f) At the second iteration, repeatedly perform (b), (c), (d) and (e) on the trailing sub-matrix that starts from the second column of tiles. (adapted from [Song et al. 2012]

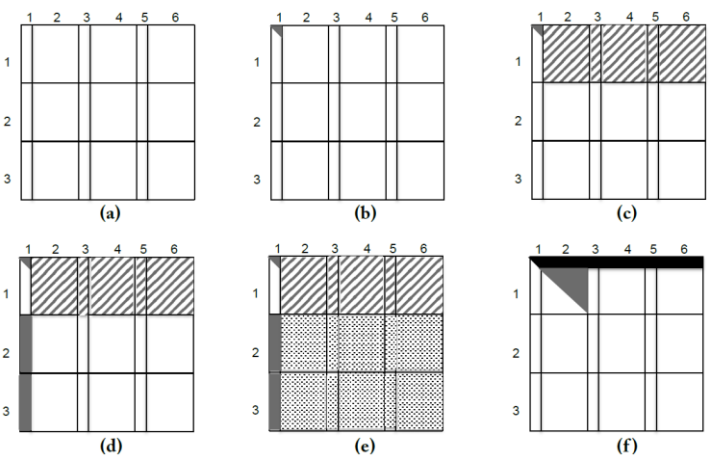


Fig. 4.5 Operations of heterogeneous tiled QR factorization. (a) The matrix *A*. (b) Compute the QR factorization of A_{11} to get R_{11} and V_{11}. (c) Apply V_{11} to update all tiles on the right of A_{11} by calling LARFB (BLAS routing to apply a block a reflector). (d) Compute TSQRT (BLAS routine to compute tall-skinny QR on a triangular matrix) for all tiles below A_{11} to solve V_{21} and V_{31}. (e) Apply SSRFB (BLAS routine) to update all tiles on V_{21} and V_{31}'s right hand side. (f) After

the 1st iteration, repeatedly performing (b), (c), (d) and (e) on the trailing sub-matrix. (adapted from [Song et al. 2012]

- **Communication-avoiding**

Techniques in this category aim to reduce the overhead of communications between CPU and GPU and/or GPU and global memory. In dense linear algebra computations, the pivoting operation is an important numerical technique to prevent divisions by very small values in the Gaussian elimination (GE) process. Many factorization methods involve a pivoting operation when processing each column of panel. The partial pivoting consists of finding the elements of highest magnitude (so-called *pivot*) within the column and then swapping the corresponding rows. Although the commonly used Gaussian Elimination methods with Partial Pivoting (GEPP) can result in algorithms with good convergence property, it introduces a large amount of communication overhead due to frequent swapping of rows. It also limits the parallelism among tiles because the update of the trailing sub-matrices can only be processed after the pivoting based panel factorization is completed. The cost of pivoting can increase the time of factorization by up to 40% on some hybrid architectures [Baboulin et al. 2012].

Baboulin et al. [2012] developed a communication-avoiding factorization algorithm to minimize data communication. In this work, the panel factorization is performed using a communication-avoiding pivoting heuristic. The panel factorization process, referred to as Tall Skinny LU (TSLU) in BLAS terminology, can be efficiently parallelized as follows. The panel is partitioned into P_r blocks. From each block, a set of local pivots is selected in parallel. A traversal is performed on the P_r sets to select a set of global pivots. These global pivots are moved to the diagonal positions, and then the LU factorization without pivoting of the entire panel is performed. The hybrid LU decomposition algorithm proposed by Baboulin et al. [2012] is built by extending the TSLU routine. Fig. 4.6 shows an example of the factorization at the top level. The initial matrix is stored on the GPU. Red tasks represent the factorization of the panel and are performed by multiple

threads on CPU, while the green tasks are the updates of the trailing sub-matrices done by a GPU. At each step of the factorization, the block corresponding to the panel is transferred to the CPU and factored. Once a panel is factored, it is transferred back to the GPU to update the trailing sub-matrix. The GPU concurrently updates the column blocks corresponding to the next panel. Therefore, the data transfer between CPU and GPU can be overlapped with computation.

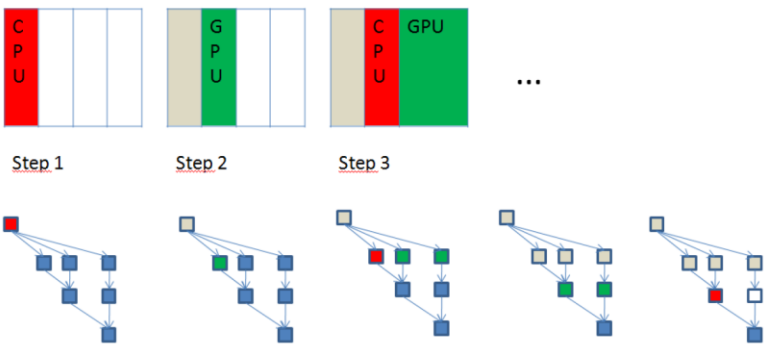


Fig.4.6 Hybrid CALU factorization with 4 panels [Baboulin et al.2012]

Khabouet et al. [2012] further optimized LU factorization with a panel rank revealing pivoting heuristic. It is proved to be more stable than GEPP in terms of worst case growth factor and robust on various classes of matrices, including pathological cases on which GEPP fails. Anderson et al. [2010] describe an implementation of communication-avoiding QR factorization and demonstrate the effective reduction in traffic between CPU and GPU on a large class of tall-skinny matrices.

- **Task Scheduling**

For heterogeneous platforms with multiple CPUs and GPUs, it is essential to map tasks to computing resources in an optimized fashion in order to maximize the overall performance. Such a task scheduling process plays a fundamental role in extracting parallelism, maintaining good load balance, and reducing communication. Generally, existing research works solve the task scheduling problem from two directions, static scheduling and dynamic scheduling [Kurzak et al. 2010]. Static scheduling techniques allocate tasks

between CPU(s) and GPU(s) in a fixed manner, while dynamic scheduling approaches assign tasks by considering the runtime information.

Early works (e.g., [Barrachina et al. 2009; Volkov and Demmel 2008]) employed a static schedule to start tasks on host CPU and device GPU(s) due to its simplicity. They take into account the heterogeneity of both processors and memories for better performance when doing dense matrix factorizations. Song et al. [2012] indicate that the static scheduling method can be faster than dynamic scheduling on relatively small matrices (e.g., 60% faster on a matrix with a dimension of 5760 on 4 CPU cores). However, static scheduling often requires an auto-tuning procedure on a set of data of varying sizes and still lacks of flexibility for optimally processing matrices of arbitrary sizes.

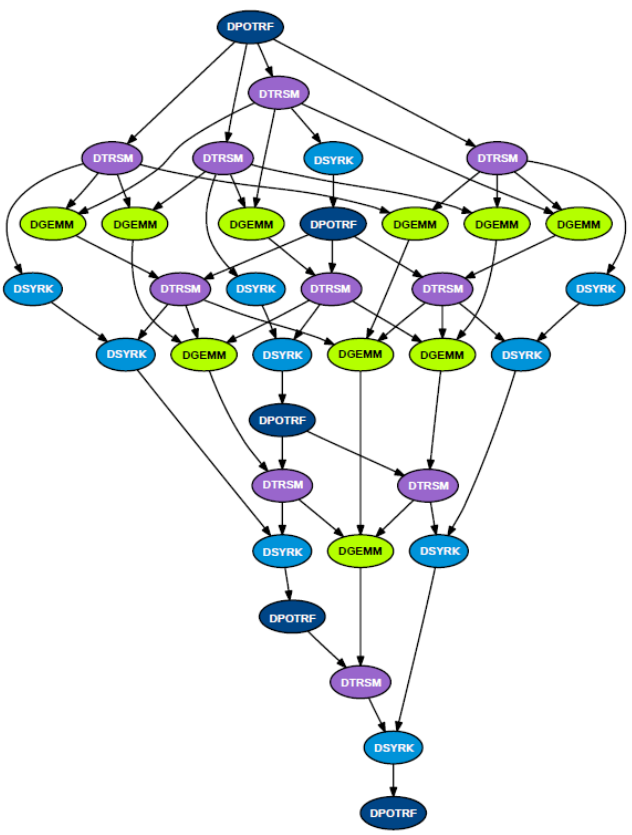
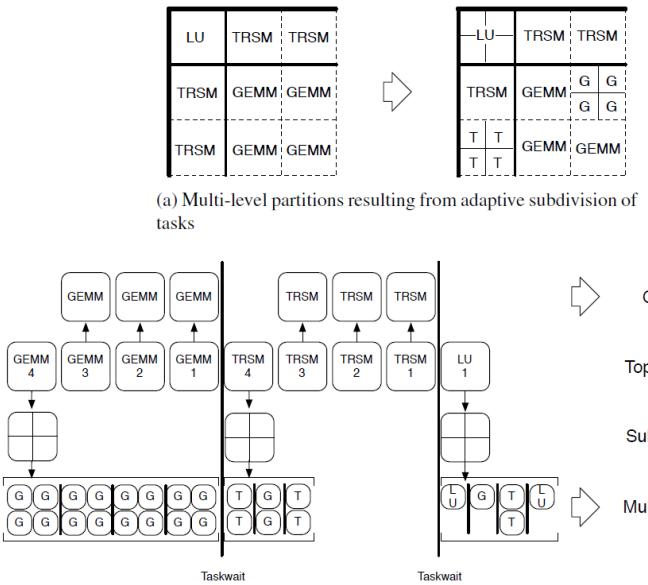


Fig. 4.7 Task Graph of the tiled Cholesky factorization [Kurzak et al. 2010]

Given a computation procedure, the tasks and dependencies are usually captured with a task graph, which is a directed acyclic graph (DAG) [Thulasiraman and Swamy 1992]. In a task graph, nodes represent computations tasks and edges stand for data dependencies among tasks. Fig. 4.7 illustrates the task graph of a tile-based Cholesky factorization algorithm. Given the dependency information delineated in the task graph, a dynamic scheduler is responsible for distributing these tasks to different computing devices by taking into account load balance, synchronization overhead and communication cost.

Agullo et al. [2011] proposed to dynamically schedule the LU factorization process on a multi-core CPU and one or more graphics cards with the StarPU runtime[11] [Augonnet et al. 2010]. StarPU automates data management on a heterogeneous platform and provides a flexible framework to design scheduling policies. Under such a context, all data have to be registered at the beginning. The LU factorization process is decomposed into a series of tasks working on blocks of the input matrix by following the tiling algorithm introduced by Buttari et al [2009]. During execution, a runtime scheduler maintains a task queue of ready tasks and dynamically assigns them to a computing node (i.e., either a CPU core or a GPU) with a greedy strategy. A history based prediction model is built inside the scheduler to estimate the completion time of a given task on a given computing node. The scheduler always picks a ready task and allocates to a computing device to maximize the predicted performance.

[11] A runtime system is a set of instructions that are inserted into the executable by a compiler. It provides certain services to an application as a set of APIs.



(a) Multi-level partitions resulting from adaptive subdivision of tasks

(b) Heterogeneous task scheduling

Fig.4.8 Workload allocation between CPU and GPU (adapted from [Kim et al. 2012])

Kim et al. [2012] developed a hybrid scheduling scheme for LU factorization. The method considers the computing power of various computing devices and identifies tasks by adapting to target devices. Fig. 4.8 depicts an example for a tile-based LU factorization without pivoting. The detailed procedure of tile-based LU factorization is shown in Fig. 4.2. In each iteration, there are three super-steps, LU factorization, GEMM operations and TRSM operations, as shown in Fig. 4.8(a). At each super-step, the matrix is divided into sub-blocks to be processed as tasks. The division is designed to maintain load balance between CPU and GPU. The scheduling details are illustrated in Fig. 4.8(b). The first LU task is directed to CPU. Then a dynamic scheduler allocates tasks between CPU and GPU. Lighter tasks of GEMM and TRSM are scheduled to CPU, while heavier ones are mapped to GPU.

4.1.2.2 Multigrid methods

Multigrid methods were originally developed to solve large PDE problems [Briggs 1987]. Later it was successfully applied to solve linear systems as one of the fastest solvers [Shapira

2009]. Multigrid methods operate on a hierarchy of grids, with the finest grid corresponding to the original problem and other grids being iteratively derived by coarsening a finer grid. The solution process repeatedly iterates through the hierarchy of grids until the convergence condition is attained. Due to the regular grid structure, multigrid methods are amenable to GPU implementations. It is straightforward to map different nodes to different threads. Nevertheless, the computations inside a node of the grid can be irregular and so the load balance is still a concern.

NVIDIA and ANSYS recently developed a highly tuned GPU based multigrid library, nvAMG [Strzodka 2012]. It parallelizes all phases of algebraic multigrid. Feng and Li [2008] proposed GPU based multigrid solution techniques for power grid analysis. A key innovation of the works is the conversion of a generally irregular problem into regular grids.

4.2 Sparse Linear Algebra

A VLSI design must be captured in a certain design representation to be processed by EDA software. In such representations, no matter at logic level or layout level, a design object (e.g., a gate or a wire) generally has interconnections with only a few neighboring objects. When using matrices to capture such interactions, it naturally follows that only a small percentage of matrix entries have a non-zero value. Accordingly, sparse matrix is one of the most fundamental data structures of EDA applications. Sparse matrix is also perhaps the most popular irregular data structure and thus of special importance to study. Excellent reviews and textbooks on parallel sparse matrix computation can be found in [Heath et al. 1991; Liu 1986; Liu 1992; Saad 2000].

Sparse matrices can be stored in many different formats. Please refer to [Saad 2000] for a thorough treatment. Among various formats, the Compressed Sparse Row (CSR) format and its column-oriented derivative, Compressed Sparse Column (CSC) format, are the most widely-used general-purpose formats of sparse matrices. Other formats like the diagonal format (DIA) and ELLPACK can be more efficient for more structured

matrices [Bell and Garland 2009]. In this survey, we focus on the CSR format and almost all techniques covered in this work can be used for the CSC format by properly exchanging the row and column indices.

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 4 & 5 & 0 & 6 \end{bmatrix} \quad \begin{aligned} rowptr &= \{1, 3, 3, 4, 7\} \\ col &= \{1, 2, 1, 1, 2, 4\} \\ elem &= \{1, 2, 3, 4, 5, 6\} \end{aligned}$$

(a) A exemplar sparse matrix (b) CSR storage for the matrix in (a)

Fig. 4.9 An example sparse matrix and its compressed sparse row storage

Fig. 4.9 exemplifies the CSR format. Given a sparse matrix *A* shown in Fig. 4.9(a), the CSR format needs 3 arrays to capture the original sparse matrix. Among the three, arrays *elem* and *col* store the non-zero values and the corresponding column indices, respectively. The values are contiguously stored in both arrays without the row index information, which is left to the third array, *rowptr*. This array has an entry for each row of the original sparse matrix. The value of *rowptr[i]* records the index in array *elem* for the 1st nonzero element of row *i* of the original sparse matrix. For instance, the example matrix *A* in Figure 4.9 has one non-zero in the first row. The non-zero is placed in the first entry of array *elem*. So *rowptr[1]*[12] has a value of 1, which means the first non-zero of row 1 appears in the first entry of *elem*. Next, the second row of matrix *A* has its first non-zero located in the 3rd entry of *elem* and *rowptr[2]* is set as 3. All elements in the 3rd row of matrix *A* are zero. Hence, *rowptr[3]* is also equal to 3. A nice feature of CSR is that the difference between *rowptr[i+1]* and *rowptr[i]* is equal to the number of non-zeros on row *i*. In fact, *rowptr[i]* and *rowptr[i+1]* defines a range in array *elem*. A traversal of array *elem* in the above range gives every non-zero in row *i* of the original sparse matrix. Array *rowptr* usually has an extra entry at the end with its value set as the index of last entry of array *elem* plus 1. Such an arrangement simplifies the access to all non-zeros in the last row.

[12] In this example, we assume that the array index starts from 1

In EDA and other engineering applications, sparse matrices can be excised by many different algebra computations. Among these, sparse matrix vector product (SMVP or SpMV) and LU factorization [Saad 2000] are generally the bottleneck of EDA applications [Deng et al. 2009]. In this section, we review recent progresses in using GPU to accelerate SMVP and LU factorization.

4.2.1 Sparse Matrix Vector Product

All sparse matrix algorithms involve a series of operations that can be classified by their operands as scalar-scalar, scalar-vector, vector-vector, matrix-vector, and matrix-matrix computations. Obviously, the matrix-vector operation has a higher complexity than the former 3, while the matrix-matrix computation can be decomposed into a set of matrix-vector operations. The SMVP computation thus plays an essential role in sparse linear algebra. A key observation from the profiling results on a group of sparse matrix based EDA applications including circuit simulation, circuit placement and finite element based layout stress analysis is that SMVP is often the bottleneck. For instance, a conjugate-gradient based linear system solver [Deng and Mu 2008] for the placement problem spends more than 90% of CPU time in computing SMVP.

Code 4.6 lists the serial C code for computing SMVP in the CSR format. Besides the 3 arrays for the input sparse matrix, we also need an array, *v*, to store the input vector and an array, *p*, to record the product vector. The number of rows of the input matrix is *num_rows*. The code consists of an outer loop and a nested inner loop. Every round of the outer loop generates one element of the product vector. The inner loop traverses all non-zero elements in one row. For the *j*th element in array *elem*, we need to extract the corresponding vector element by first locating the column index *col[j]*.

Code 4.6 C code for computing sparse matrix vector product

```
void smvp_serial(const int *rowptr, const int *col,
  const float *elem, const int num_rows, float *v, float *p)
{
  for(int row=0; row<num_rows; ++row) {
     int row_begin = rowptr[row];
     int row_end = rowptr[row+1];
     float sum = 0.0;
     for(int j= row_begin; j<row_end; ++j)
        sum += elem[j] * v[col[j]];
     p[row] = sum;
  }
}
```

Algorithm 4.2 Data-parallel row-based SMVP

input: Sparse matrix *A* in 3 arrays *rowptr*, *col*, and *elem*, vector *v*
output: Product vector *p*

```
begin
   for each row of A in parallel do
      row_begin = rowptr[row]
      row_end = rowptr[row+1]
      sum = 0.0
      for all j from row_begin to row_end-1 do
         sum = sum + elem[j] * v[col[j]]
      end for
      p[row] = sum
   end for each
end
```

Code 4.7 Row-wise parallel CUDA code for SMVP [Deng 2009]

```
1  __global smvp_row_parallel(const int num_rows, cont int *rowptr,
2          const int *col, const float *elem, const float *v, float *p)
3  {
4      int row = blockIdx.x*blockDim.x + threadIdx.x;
5      if( row<num_rows ){
6          int row_begin = rowptr[row];
7          int row_end = rowptr[row+1];
8          float sum = 0.0;
9          for(int j= row_begin; j<row_end; ++j)
10         sum += elem[j] * v[col[j]];
11         p[row] = sum;
12     }
13 }
```

Starting with the code listed in **Code** 4.6, a straightforward parallel implementation of SMVP on GPU is to create as many

threads as number of rows. Every thread computes an inner product of a single row and the input vector. The corresponding parallel algorithm is listed as **Algorithm** 4.2. References [Bell and Garland 2009; Garland 2008] provide details for such an approach. The corresponding CUDA code snippet is listed in **Code** 4.7.

The performance of **Code** 4.7, however, turns out to be unsatisfying. A careful analysis on **Code** 4.7 reveals two major reasons leading to significant inefficiency. First, the memory accesses generally cannot be fully coalesced. Given a warp of threads, the memory addresses incurred by lines 6 are not contiguous. For earlier GPUs (prior to Fermi GPU), such access patterns cannot be coalesced at all. The problem becomes less serious on Fermi and later GPUs. The same also happens to Line 7. The problem of Line 10 is even more complicated. The percentage of memory requests to array *elem* that can be coalesced is determined by the distribution to elements. In addition, it is infeasible to coalesce all accesses to array *v* due to the indirect addressing in the form of *v[col[j]]*. Secondly, the load balance can be poor. A thread has to traverse every element in one row. However, the distribution of non-zeros in rows can be significantly uneven. In a typical adjacent matrix of IC designs, most rows have only a few non-zeros, while there are a few rows (e.g., those correspond to clocks) consisting of hundreds or even thousands of nonzero elements. As a result, threads in one warp may have significantly varying workload, which is especially inefficient on GPUs. Therefore, the GPU run time is dominated by those less sparse rows. The above problems make SMVP problem extremely challenging for GPUs. There exist tough matrix instances where the straightforward GPU implementation can be slower than its CPU equivalent [Deng and Mu 2008].

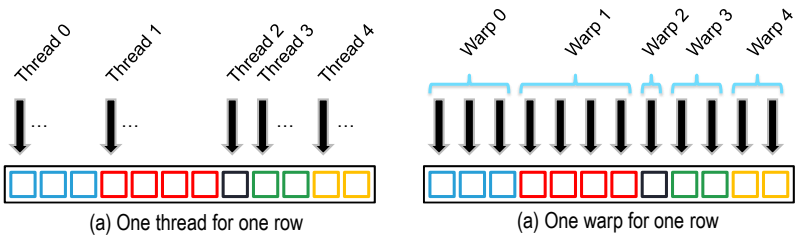


Fig. 4.10 Comparison of **Code** 4.7 and **Code** 4.8

Bell and Garland [2008; 2009] proposed a novel solution to the CSR based SMVP problem. This work uses a warp, i.e. 32 threads on Fermi GPU, to process one row. The corresponding code is listed in **Code** 4.8. The underlying parallel work organizations of **Code** 4.7 and 4.8 are compared in Fig. 4.10.

In **Code** 4.8, a warp of 32 threads first fetches data from array *elem* (Line 17-19). The fetched data is directly multiplied with the corresponding vector element. Since there may be over 32 elements in one row, a summation is needed when a thread iterates the elements. Now the visits to array *elem* can be fully coalesced because neighboring threads in a warp fetch contiguous memory addresses into the shared memory. Such a technique of collaborative accessing global memory by a warp is a commonly used and extremely effective micro-pattern in optimizing GPU programs. After executing Lines 17-19, 32 partial sums are derived. The next step is to perform a summation of the 32 values. It follows the "reduce" pattern of data-parallel programming [Blelloch 1990]. The code on lines 22-26 is for this purpose. The idea is to first use one half of the threads to calculate a summation for every two elements. Then one fourth of the threads perform another summation for every two partial sums derived in the previous step. The process continues until there is only one element remains. Note that there is no need for explicit synchronization after each summation because threads in one warp are automatically synchronized (i.e., always complete an identical instruction in one instruction cycle). The processing flow for 8 data elements is illustrated in Fig. 4.11 [Harris 2008], where the arrows represent threads in a warp and the rectangles stand for memory addresses storing the intermediate and final sums.

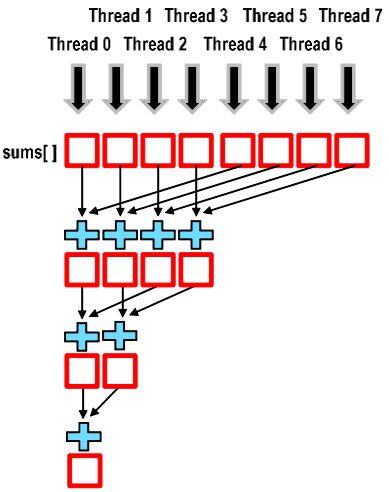


Fig. 4.11 Intra-warp reduction based summation

The advantages of the above approach are two-fold. First, memory accesses to array *elem* can be coalesced because the 32 threads in one warp work together to fetch the non-zeros in one row. Second, the load balance among threads in one warp is not a problem anymore. The resultant GPU implementation is extremely efficient for sparse matrices with a few tens of or more non-zeros in most rows. It is the first research result to deliver a throughput of over 10 GFLOPS on commonly used sparse matrix benchmarks. Such a method was later adopted by the CUSPARSE package [Naumov et al. 2010].

Code 4.8 Warp-based CUDA code for SMVP [Bell and Garland 2008]

```
1__global__ void spmv_csr_vector_kernel ( const int num_rows ,
2         const int * rowptr, const int * col, const float * elem ,
3         const float *v, float * y)
4{
5   __shared__ float sums [];
6   int thread_id = blockDim.x * blockIdx.x + threadIdx.x ;
7   int warp_id = thread_id / 32; // global warp index
8   int lane = thread_id& (32 - 1); // thread index within the warp
9
10   // one warp per row
11   int row = warp_id ;
12   if ( row < num_rows ){
13         int row_start = rowptr [row ];
14         int row_end =rowptr [ row +1];
15
16         // compute running sum per thread
17         sums[ threadIdx.x ] = 0;
18         for ( int j = row_start + lane ; j<row_end ; j += 32)
19             sums[ threadIdx.x ] += elem[j] * v[ col[j]];
20
21         // parallel reduction in shared memory
22         if ( lane < 16) sums[ threadIdx.x ] += sums[ threadIdx.x + 16];
23         if ( lane < 8) sums[ threadIdx.x ] += sums[ threadIdx.x + 8];
24         if ( lane < 4) sums[ threadIdx.x ] += sums[ threadIdx.x + 4];
25         if ( lane < 2) sums[ threadIdx.x ] += sums[ threadIdx.x + 2];
26         if ( lane < 1) sums[ threadIdx.x ] += sums[ threadIdx.x + 1];
27
28         // first thread writes the result
29         if ( lane == 0)
30             p[ row ] += sums[ threadIdx.x ];
31    }
32}
```

Nevertheless, the implementation is less efficient for problem instances arising from EDA applications, where most rows have only a limited number of non-zeros [Deng and Mu 2008]. Under such a situation, threads inside a warp cannot be fully exploited and thus the GPU hardware is underutilized. In addition, the indirect memory addressing in line 19 of **Code** 4.8 still incurs un-coalesced memory accesses.

To address the abovementioned problems, Deng et al. [2009] proposed a GPU based SMVP procedure, which is orthogonal to the approach developed by Bell and Garland [2009]. A basic

observation on **Code** 4.6 is that the SMVP computation actually consists of two phases with different available parallelism. In the first phase, every non-zero matrix element must be multiplied by a corresponding vector element. From this point of view, the multiplication operations are fully regular and predictable. In the second phase, the sum of the products needs to be calculated for each row. Here the number of summations per row is determined by the distributions of non-zeros and thus cannot be regular for general cases. The two phases can be organized as two GPU kernels executed sequentially.

Code 4.9 Straightforward implementation of the product kernel [Deng et al. 2009]

```
1 __global__ void product_kernel(const int *col, const float *elem,
2                     const int num_nz, const float *v, float *middle)
3 {
4     int elemid = blockIdx.x * blockDim.x + threadIdx.x;
5     if( elemid <num_nz)
6         middle[elemid] = elem[elemid] * v[col[elemid]];
7 }
```

The first kernel, designated as the product kernel, can be implemented in a straightforward manner by assigning one multiplication to each thread. The corresponding code is listed in **Code** 4.9. The product of every pair of matrix and vector elements is stored in array *middle*. Note that the computation loads are also perfectly balanced. Here the difficulty lies in the fetching of vector elements from the global memory. As shown in line 6 of **Code** 4.9, the load operation, i.e. accessing *v[col[elemid]]*, generally cannot be coalesced because the value of *col[elemid]* can be arbitrary. An expansion operation as illustrated in Fig. 4.12 can be applied to vector *v* to improve the performance. The expanded vector, *v_expanded*, has the same length as array *elem* with *v_expanded[elemid]* = *v[col[elemid]]*. The expansion operation can be efficiently implemented by loading arrays *v* and *col* into the shared memory and then write *v_expanded* according to the mapping

relations[13]. Experiments indicated that it takes less than 20% of the execution of the product kernel [Deng et al. 2009]. In addition, many applications allow the expanded vector to be reused for many times. It should be noted the high sparsity of EDA matrices makes the expansion beneficial. With the expanded vector, memory coalescing can be perfectly achieved. **Code** 4.10 lists the improved implementation with completely coalesced memory access.

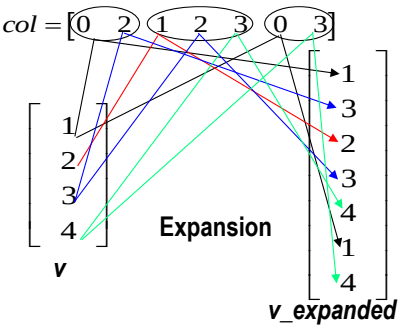


Fig. 4.12 Example of vector expansion [Deng et al. 2009]

Code 4.10 Implementation of the product kernel with expanded vector [Deng et al. 2009]

```
1 _global__ void expanded_product_kernel(const float *elem,
2           const int num_nz, const float *v_expanded, float *middle)
3 {
4     int elemid = blockIdx.x * blockDim.x + threadIdx.x;
5     if( elemid <num_nz)
6           middle[ elemid] = elem[ elemid] * v_expanded[ elemid];
7 }
```

The second kernel, which is called the summation kernel, is responsible of summing the products in one row together and then exporting it to the product vector. The major bottleneck is again the un-coalesced memory accesses due to the irregular distribution of non-zeros.

[13]Arrays *v* and *col* are generally bigger than the capacity of the shared memory on a single multiprocessor. So each multiprocessor can only load a sub-array of *v* and *col*, respectively. If an element required by column index is outside the shared memory, then additional global memory loads are still needed.

A certain capacity (e.g., configured to be 16KB or 48KB on Fermi GPU) of shared memory is installed into each multiprocessor of GPUs. In the case of SMVP, each product created by the first kernel is only used once in the summation process and so there is no need for data sharing among different threads. However, the shared memory can be used to make the memory access coalesced. The idea is illustrated in Fig.4.13, where GMEM stands for global memory and SMEM represents shared memory. Threads in one block work together to load the products created by the product kernel into shared memory in a coalesced manner. The summation process then uses data in the shared memory. Of course, due to the size limit of shared memory, not all data can be cached. Deng et al. [2009] developed a CPU based shared memory simulator to evaluate the effectiveness of the above technique. Given 768 threads running on one multiprocessor, every thread loads 5 or 6 floating point numbers (4 bytes each) into the shared memory. The results prove that the "hit ratio" (probability of required data in the shared memory) can be higher than 90% for a wide range of matrix instances when the average number of non-zeros per row is less than 6 for a 16KB shared memory.

The summation kernel is listed in **Code** 4.11. The code of lines 8-9 derives the index of the elements required by the 1st thread in the current thread block. The index is then aligned to be a multiple of 16 as required by the coalescing rules. The code on lines 12-17 coordinates threads in one block to work together to load data into shared memory. The remaining code performs the summation. Before an adding operation, the code on lines 26-27 checks if the data is already cached in the shared memory. If it is not cached, a global memory access is still needed.

The techniques proposed by Deng et al. [2009] were tested on a large number of highly sparse matrices arisen from various EDA applications. The GPU implementation outperforms the results of Bell and Garland [2009] by a factor or 5 to 10. Note that the summation kernel can also be realized by the warp based technique developed by Bell and Garland [2009].

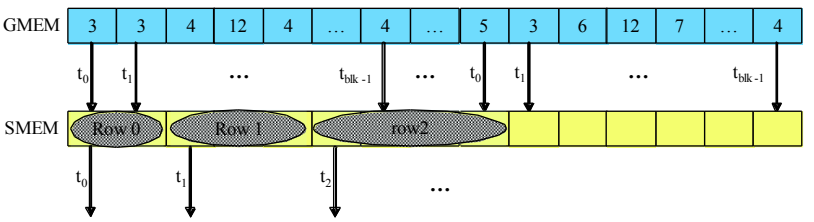


Fig. 4.13 Coalescing through shared memory (GMEM represents GPU global memory and SMEM stands for shared memory) [Deng et al. 2009]

Code 4.11 Source code of summation kernel [Deng et al. 2009]

```
__global__ void summation_kernel(const int *rowptr, float *middle,
        const int num_row, const unsigned num_nz,
        const int num_loads_per_thread, float *p)
{
   __shared__ float cache[SMEM_PER_BLK];
   __shared__ int num_nz_before;
   int thread_begin = blockIdx.x * blockDim.x;
   if(threadIdx.x == 0)
        num_nz_before = rowptr[thread_begin]/16*16;
    __syncthreads();
   int elemid, cache_idx;
   for( int i = 0; i <num_loads_per_thread; i++){
        cache_idx = i * NUM_THREAD_PER_BLOCK + threadIdx.x;
        elemid = num_nz_before + cache_idx;
        if( cache_idx < S SMEM_PER_BLK && elemid <num_nz)
        cache[cache_idx] = middle[elemid];
   }
  __syncthreads();

   int row = thread_begin + threadIdx.x;
   if ( row <num_row){
        float sum = 0.0;
        int row_begin = rowptr[row];
        int row_end = rowptr[row + 1];
        for (int i = row_begin; i <row_end; i++){
        if( i >= num_nz_before && (cache_idx = i – num_nz_before) <
                SMEM_PER_BLK)
                sum += cache[ cache_idx];
        else
                sum += middle[ i];
        }
        p[row] = sum;
   }
}
```

The throughput of GPU on sparse matrices now reaches 10-20 GFLOPS, but is still substantially lower than the peak

throughput of GPUs. Accordingly, the SMVP problem continue to attract considerable research efforts. Such works can be categorized into the following directions.

- **Automatic tuning**

One essential problem is that different sparse matrices require varying configurations of the SMVP kernel and storage format for optimal performance. Choi et al. [2010] developed a model-driven approach for automatic tuning of GPU based SMVP code. An analytical performance model is proposed by considering both kernel configuration (i.e., block and grid dimension) and hardware resources (e.g., number of registers and capacity of shared memory). Various parameters in the model like the initialization cost have to be identified through off-line profiling of real executions of micro-benchmarks. Given an input matrix, the auto-tuning framework determines an optimized configuration for SMVP code. Experimental results show that the resultant performance is within 15% of that can be achieved by exhaustive search. A similar approach was proposed in [Guo et al. 2011]. Matam and Kothapalli [2011] developed a hybrid technique by organizing code for different sparse matrix formats into a single kernel. The underlying observation is that the ELLPACK format [Grimes et al. 1979] is likely to outperform the CSR format on rows with less than 256 non-zeros. The allocation of rows among two formats is determined before kernel launching.

- **Preprocessing**

Reordering has been an important technique to improve the data locality of sparse matrices and thus the performance of SMVP. The idea is to permute the rows or columns so that a matrix has a better distribution of non-zero elements. Buatois et al. [2009] investigated the Reverse Cuthill McKee reordering heuristic. Pichel et al. [2012] developed a detailed investigation on the impact of various reordering heuristics on GPU based SMVP computation. This work proves the effectiveness of reordering and the most noticeable speed-up by reordering alone can be up to 2.6X. In addition, distance

function [Pichel et al. 2008] based reordering seems to be the most promising reordering heuristic.

- **Optimization**

Even with an efficient algorithm, careful coding and intensive optimization have to be performed to attain a satisfying level of performance on GPUs. Baskaran and Bordawekar [2009] proposed a set of code optimization techniques for the SMVP problem. Implementing high performance GPU code is a labor-intensive and error-prone process. To ease the effort, Grewe and Lokhmotov [2011] introduced a code generation and tuning framework, which produced CUDA code from a high-level representation and performed optimizations in an automatic manner.

4.2.2 Iterative Solver

Given a linear system $Ax=b$, where A is typically a sparse matrix in EDA applications and b is a known vector, a solver finds the solution to the unknown vector x. EDA and other engineering problems often need to solve large linear systems. Linear system solvers can be grouped into 2 categories, iterative solvers and direct solvers. Generally, iterative solvers are faster, but there are also important applications in which direct solvers have a performance advantage. In this sub-section, we review GPU accelerated iterative solvers.

Iterative linear system solvers have extensive applications in EDA applications and are also often the performance bottleneck. One typical application is the analytical placement (e.g. [Kleinhans et al. 1991; Eisenmann and Johannes 1998]), where more than 60% of the CPU time is spent on solving linear systems using a Conjugate Gradient (CG) solver [Barret et al. 1994]. A CG solver has a processing flow as shown in **Algorithm** 4.3. Other solution algorithms are based on the same set of computing patterns but organized in different flows.

Algorithm 4.3 involves a series of vector and matrix operations. Among these, the SMVP kernel is usually the most time-consuming building block. In a force-driven placer [Deng and Mu 2008], over 90% CPU time of a CG solver is

consumed by the SMVP procedure. The techniques presented in Section 4.2.1, therefore, can be directly applied to accelerate iterative solvers on GPUs. Other vector operations can be implemented very efficiently on GPU. The GPU implementations of such kernels as SAXPY (sum of Alpha × x + y, where Alpha is a scalar and x and y are vectors), inner product, and reduction can outperform their CPU counterparts by 50-100 fold. A fully GPU accelerated CG solver delivered a 20X speed-up [Deng et al. 2009]. Other works also presented encouraging results on using GPU to accelerate the solution of linear systems (e.g., [Jost et al. 2009; Helfenstein and Koko 2012]). Note that the CG method can also be used to solve nonlinear optimization problems. Galiano and Koko [2012] reported results of applying GPU in nonlinear optimization.

Algorithm 4.3 Preconditioned CG method
input: Ax=b, where A tend to be sparse in EDA applications, and a precondition matrix M
output: Solution to unknown vector x
begin
  Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
  for i = 1, 2, . . .
    solve $Mz^{(i-1)} = r^{(i-1)}$ //permutation or vector operations
    $\rho_{i-1} = (r^{(i-1)})^T z^{(i-1)}$ //inner product
    if i = 1
      $p^{(1)} = z^{(0)}$ //scalar operation
    else
      $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$ //scalar operation
      $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$ //vector scaling and addition
    end if
    $q^{(i)} = Ap^{(i)}$ //sparse matrix and vector product
    $\alpha_i = \rho_{i-1}/(p^{(i)})^T q^{(i)}$ //inner product and scalar
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ //vector scaling and addition
    $r^{(i)} = r(i-1) - \alpha_i q^{(i)}$ //vector scaling and subtraction
    check convergence; continue if necessary
  end for

4.2.3 LU Factorization and Direct Method for Solving Linear System

The direct method is another important class of linear system solvers. Originated from the Gaussian elimination method, the

idea is to decompose a matrix into triangular matrices by LU, Cholesky, or other factorization methods. Then the solution of the original problem can be derived by continuously solving two triangular linear matrices. Direct solvers are generally slower than their iterative equivalents. However, if the factorization pattern only needs to be solved once and many triangular iterations can reuse the same factorization pattern, the direct method may have a considerable performance advantage. As a result, SPICE-alike circuit simulators pervasively adopt the direct method in their solvers. In this sub-section, we review GPU accelerated LU factorization, but the techniques also apply to other factorization methods.

Sparse matrices pose special challenges to the direct method. The major problem is that row and/or column information is compressed in sparse matrix formats and thus extra efforts have to be made to extract the information. The resultant data dependency significantly complicates the computation process. In addition, the factorization process introduces new non-zero entries into a sparse matrix. Re-constructing a sparse matrix is generally expensive in CPU time.

EDA vendors have released successful multi-core based direct solvers in their circuit simulation solutions (e.g., [Cadence 2008; Synopsys 2008a]). On the other hand, few works address the problem of using GPU to accelerate sparse direct solvers. This is partially due to the inefficiency of GPU to execute highly data dependent code, and partially because of the relative success in parallelizing circuit simulators on multi-core CPU platforms.

One major contribution along this direction is reported in [Ren et al. 2012]. This work uses a hybrid flow depending on both CPU and GPU to perform LU factorization. The overall flow is demonstrated in Fig. 4.14. A series of preprocessing steps are applied to the input matrix. Such steps include reordering to reduce fill-ins and deriving the symbolic factorization so that the non-zero patterns can be determined. The preprocessing operations are carried on CPU only once. The succeeding calculation of the entries of the L and U matrices are performed on GPU in an iterative manner.

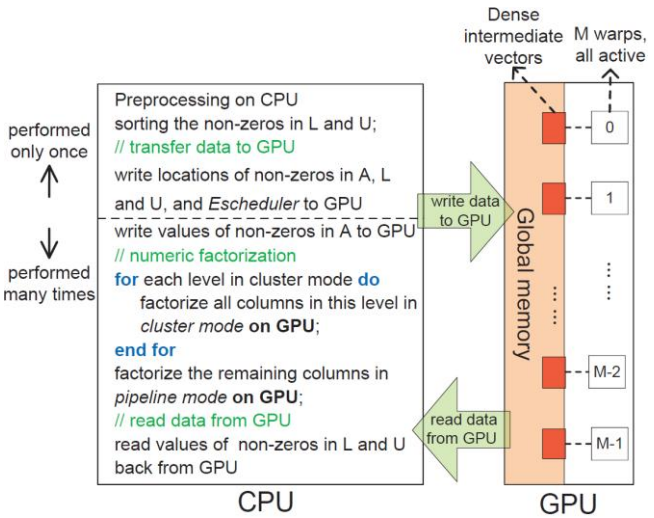


Fig. 4.14 The Processing flow of GPU accelerated sparse LU factorization proposed by Ren et al. [2012]

The computation of L and U matrix is based on the G/P left-looking algorithm [Gilbert and Peierls 1988]. The idea is to process the matrix in a column-by-column manner from left to right. At each iteration, the current column is first updated by the entries from proceeding columns, and then processed with column specific operations. Fig. 4.15 illustrates an exemplar computation process, where circles represent columns and arrows stand for computation operations. The dotted curves correspond to steps of processing.

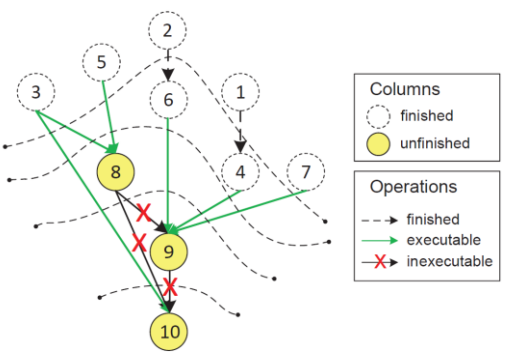


Fig. 4.15 Parallelism in sparse LU factorization [Ren et al. 2012]

Ren et al. [2012] exploited two levels of parallelism in the left-looking algorithm. The lower level parallelism is the vector

operation inside a column and one warp of threads are assigned to one column. The higher level parallelism lies among different columns without dependencies. Due to the dynamic nature of such parallelism, all necessary threads for processing a matrix have to reside on a GPU from the beginning of computation so that they can start an operation as soon as it is allowed by the data dependency relations. If there are thread blocks that cannot be assigned to GPU during kernel launch, deadlock is likely to happen. The reason is that active threads running on GPU may wait for the unassigned threads, which cannot start until the active ones finish execution and release the GPU hardware.

The work by Ren et al. [2012] offers an innovative way in exposing parallelism among complex data dependencies. Experimental results prove that the resultant GPU implementation outperforms an 8-core CPU implementation by 49% in terms of computing throughput. However, the requirement for all threads to reside on a GPU limits the scalability of this work.

4.3 Graph Algorithms

Graph algorithms constitute another major family of fundamental computing patterns that are prevalent in EDA applications. Such algorithms can be classified into two major categories, graph traversal and graph manipulation. Graph traversal algorithms, or graph search algorithms, visit every node and/or edge according to a certain order. Typical problems in this category include breadth first traversal and depth first traversal. The single source shortest path problem and various graph coloring problems also fall into this category. Graph manipulation algorithms perform certain operations on nodes and edges of a graph to optimize a given cost function. One typical problem in this class is graph partition, which is the basis of partition based placement [Caldwell et al. 2000]. Other problems include building minimum spanning tree and identifying connected components.

Many EDA applications use directed graphs. Accordingly, in this work we focus on algorithms for directed graphs, but the

fundamental techniques are generic and can also be applied to undirected graphs. For GPU processing, a directed graph is usually stored in the adjacency list as shown in Fig. 4.16. The format is similar to the CSR format for sparse matrices. Two arrays are needed for a graph. The first array, E_a, stores the outgoing edges for each node[14]. Given a node, actually only the index of every end vertex of an edge needs to be recorded. The other array, V_a, has an entry for each node. Every node has a corresponding entry in V_a to store the starting index of its outgoing edges in array Ea.

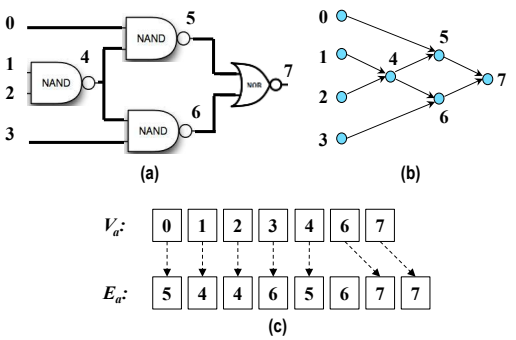


Fig. 4.16 Adjacency list representation for graph

Code 4.12 Code skeleton for GPU accelerated solutions for graph problems [Harish et al. 2009]

```
1 Create and initialize working arrays on GPU device.
2 while true do
3     for each vertex/edge/color in parallel do
4         invoke Kernel1
5     end for each
6     synchronize
7     for each vertex/edge/color in parallel do
8         Invoke Kernel2
9     end for each
10    synchronize
11
12    check termination condition; continue if necessary
13 end while
```

[14] In this paper, node and vertex are used interchangeably.

Graphs, especially those found in EDA applications, tend to be very irregular. Such irregularity poses substantial challenges for GPU acceleration. Careful tuning has to be performed to deliver a satisfying level of performance. In a pioneering work, Harish and Narayanan [2007] proposed GPU accelerated solutions to many of the classical graph theory problems (the refined versions of their solutions can be found in a succeeding work by Harish et al. [2009]). A code skeleton for these GPU accelerated solutions to graph problems is shown in Code 4.12. Such a problem typically needs to be solved in an iterative manner. Each iteration launches a few kernels sequentially. The succeeding works also follow such an organization in their coding implementations. In the remaining of this section, we review the works for different graph theory problems.

4.3.1 Breadth First Search

Breadth First Search (BFS) and its derivatives have found wide usages in EDA applications like block based timing analysis [Sapatnekar 2004], logic simulation and routing. It is also one of the most important tools for graph mining applications such as social network and genome analysis [Graph500 2010]. As a result, the problem recently attracts extensive research efforts (e.g., [Hong et al. 2011; Merrill et al. 2012; Beamer et al. 2012]).

To the best of the authors' knowledge, the first work accelerating BFS with the GPGPU computing model was proposed by Harish and Narayanan [2007]. The basic idea is illustrated in Fig. 4.17. The GPU based solution consists of two kernels organized in a loop. The parallel algorithmic flow is shown in **Algorithms** 4.4 and 4.5. The algorithm is straightforward. A thread processes a node in the graph. The nodes that are visited for the first time constitute a BFS frontier. Only nodes on the frontier need to perform certain actions. The neighboring nodes are extracted from arrays V_a and E_a. These nodes, if not visited before, will be put on the frontier for the next iteration. After the above processing, the current node is removed from the frontier. In the CPU code, the frontier is usually implemented with a FIFO queue, which needs costly

lock/unclock operations as concurrent threads may access the FIFO simultaneously. To remove the synchronization overhead, **Algorithms** 4.4 and 4.5 use two auxiliary Boolean arrays, *front* and *visit*, to indicate if a node is on the frontier of traversal and has been visited before, respectively. Note that directly updating the two arrays in **Algorithm** 4.4 may lead to race during parallel execution. Accordingly, the updating has to be done in another kernel so that a global synchronization is automatically realized through the kernel launching mechanism of GPU.

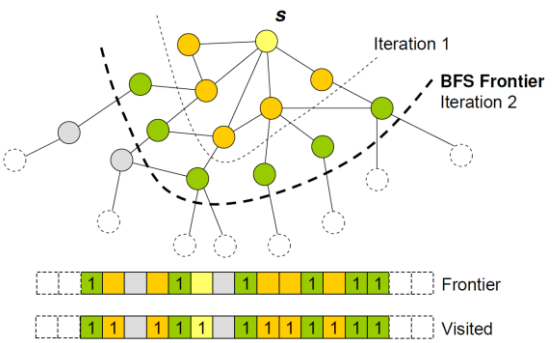


Fig. 4.17 BFS and auxiliary data structures [Harish et al. 2009]

Algorithm 4.4 Kernel 1 of GPU based BFS [Harish and Narayanan 2007]

```
input: Node array Va, edge array Ea
output: Cost array C, where C[i] records the distance of node i to root
begin
    tid = getthreadID          //one thread for each node
    if front[tid] then
          front[tid]= false
          for all neighbors nid in Ea[ Va[tid]: Va[tid+1]] do
              if NOT visit[nid] then
                  C[nid]=C[tid]+1
                  front[nid]=true
              end if
          end for
    end if
end
```

Algorithm 4.5 Kernel 2 of GPU based BFS [Harish and Narayanan 2007]

```
begin
    tid = getthreadID          //one thread for each node
    if front[tid] then
        front[tid]= true
        visit[tid] =true
        front[tid] =false
        Terminate condition is false
    end if
end
```

Note that **Algorithms** 4.4 and 4.5 are not work-efficient because every node has to be visited during each expansion of the frontier. On the other hand, an optimal sequential algorithm only visits nodes on the frontier in one iteration. Therefore, the above works perform more work than its sequential equivalent does. Such parallel algorithms are work-inefficient in the parallel algorithmic theory [Keller et al. 2001]. Such algorithms can be inefficient on sparse graphs originated from EDA applications [Deng et al. 2009]. An improved approach was proposed by Hong et al. [2011]. It maps each node to a unique warp and exploits the different threads in a warp to perform expansion.

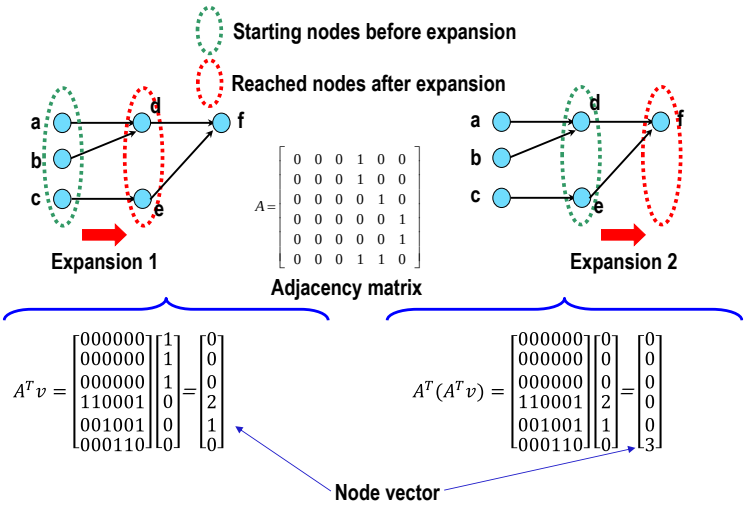


Fig. 4.18 SMVP based BFS vector product

A GPU based BFS method by extending the SMVP was proposed in [Deng et al. 2009]. The idea is to use the adjacency matrix to represent a direct graph. An entry *(i, j)* is equal to 1 if node *j* has a directed edge toward node *i*. Fig. 4.18 illustrates two expansions of breadth first traversal by starting from the primary inputs. The corresponding process can be implemented with the SMVP kernel. Given an adjacency matrix A and a vector v in which every node has an corresponding entry and the value is 1 if it is on the frontier, the product vector of $A^T v$ has one unique entry for one vertex and a non-zero entry indicates that the corresponding vertex has been reached. This procedure can be repeated until all nodes have been visited. A nice feature of this approach is the value of an entry in the product vector reflects how many paths lead to the corresponding node. A parallel reduction technique [Blelloch 1990] can be used to check if all nodes have been reached. The method proved to be efficient on a set of ATPG benchmarks. However, again the method is not work-efficient.

One early work pursuing a work-efficient BFS method on GPU was proposed by Luo et al. [2010]. It deploys a thread for each node in the current frontier to perform expansion. A thread creates a local frontier. The multiple frontiers are merged into a global frontier. This work uses a global synchronization

method developed by Xiao and Feng [2010] to coordinate the different frontiers. The synchronization processes do not involve costly launching of kernels when the number of threads to be synchronized is under a certain threshold. Instead, it depends on atomically writing to a global variable by all thread blocks that are active on all multiprocessors. This work delivers results that are able to consistently outperform an optimized CPU counterpart.

Merrill et al. [2002] introduced a novel work-efficient GPU accelerated BFS technique. This approach exploits the segmented prefix scan primitive [Blelloch 1990], which has highly efficient implementation on GPU [Sengupta et al. 2007], to perform the BFS expansion. The problem of segmented prefix scan is illustrated in Fig. 4.19. Suppose we have a vector that is partitioned into multiple segments. Then for each entry, the segmented prefix scan primitive computes the summation (or any other "reduce" operations) of the current element with all preceding entries in the same segment. If all entries belong to a single segment, the problem is called full prefix scan. Based on effective GPU implementations of full prefix scan [Harris 2007; Dotsenko et al. 2008], Sengupta et al. [2007] developed efficient solutions for the more general segmented prefix scan problem. The work by Merrill et al. [2012] integrates the prefix scan and the warp based expansion techniques. The whole procedure is organized into a hybrid flow to treat large and small expansion fronts in different manners. The overall performance is encouraging: a speed-up of 4.2x can be achieved against a respective parallel CPU implementation on 8 cores.

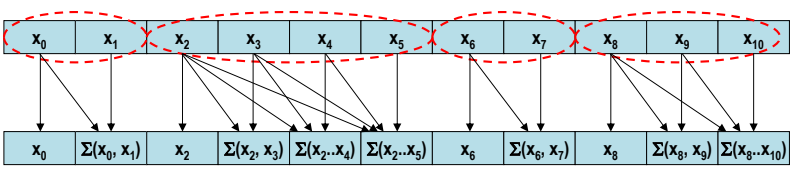


Fig. 4.19 Segmented prefix scan

4.3.2 Shortest Path

As one of the core graph theory problems, finding the shortest path in a graph is of special importance in EDA applications, especially routing and timing analysis. Besides the adjacency list, the data structures for such problems needs another array storing the weight (e.g., corresponding to length in routing and delay in timing analysis) of each edge. The shortest problem can be classified into 2 categories, single source shortest path (SSSP) and all pair shortest path (APSP). In this sub-section, we review GPU accelerated solutions to these two categories of problems.

To our knowledge, Harish and Narayanan [2007] developed the first GPU accelerated SSSP solution based on the classical Dijkstra algorithm [Cormen et al. 2001]. The corresponding data-parallel algorithms are listed in **Algorithms** 4.6 and 4.7. The basic idea is to concurrently update the distances of various nodes from a single node. The Dijkstra algorithm requires a relax process to update the path cost when a shorter path is identified. When multiple paths lead to the same node, each single update has to be performed as an atomic operation (i.e., enclosed in the "atomic" and "end atomic" pair in **Algorithm** 4.6). Otherwise, data race will happen. Again two kernels are used for global synchronization. The first kernel records the new distance values into an intermediate array, *ID*, and then the second kernel really updates the distance values. In the GPU implementations of both kernels, each thread processes one node. The performance of this method suffers from the expensive operations of atomic memory write and global synchronization.

Algorithm 4.6 Kernel 1 of GPU based SSSP [Harish and Narayanan 2007]
input: Node array V_a, edge array E_a, edge weight array W_a storing the distance between two end nodes for each edge
output: Intermediate distance array *ID*, where *ID[i]* records the intermediate cost of the shortest path from root to node *i*

```
begin
    tid = getthreadID          //one thread for each node
    if front[tid] then
          front[tid]= false
          for all neighbors nid in Ea[ Va[tid]: Va[tid+1]] do
              atomic
                    if D[nid] >D[tid] +Wa[nid] then
                          D[nid] =D[tid]+Wa[nid]
                    end if
              end atomic
          end for
    end if
end
```

Algorithm 4.7 Kernel 2 of GPU based SSSP [Harish and Narayanan 2007]
input: Node arrays V_a, edge array E_a edge weight array *Wa*
output: Path cost array *D*, where *D[i]* records the cost of the shortest path from root to node *i*

```
begin
     tid = getthreadID        //one thread for each node
    if D[tid] >ID[tid] then
          D[tid] =ID[tid]
          front[tid] =true
          Terminate condition is false
    end if
    ID[tid] =D[tid]
end
```

Algorithm 4.8 Δ-stepping algorithm for SSSP [Meyer and Sanders 1998]

input: Nodes *V*, edges *E*, edge weights *W*, where *W(i,j)* storing the weight of edge *(i,j)*, i.e. the distance between nodes *i* and *j*

output: Path cost array *D*, where *D[i]* records the cost of the shortest path from root to node *i*

```
begin
    for each v ∈V in parallel do
        heavy(v)  = {(v,w) ∈ E : W(v,w) >Δ}
        light(v)  = {(v,w) ∈E: W(v,w)≤Δ}
        d(v) =∞
    end for each
    relax(s,0)
    i =0
    while B is not empty do
        S = Φ
        while B[i] ≠Φ do
            Req= {(w, d(v) + W(v,w)) : v ∈ B[i] ∧ (v,w) ∈ light(v)}
            S = S ∪ B[i]
            B[i] = Φ
            foreach (v, x) ∈Reqin parallel do
                  relax(v, x)
            end foreach
        end while
        Req= {(w, d(v) + W(v,w)) : v 2 S ∧ (v,w) ∈ heavy(v)};
        for each (v, x) ∈Req in parallel do
            relax(v, x)
        end for each
        i = i + 1
     end while
end
procedure relax(v, x):
    if x < d(v) then
        B[⌊d(v)/Δ⌋] = B[⌊d(v)/Δ⌋]\{v}
        B[⌊x/Δ⌋] = B[⌊x/Δc⌋]∪ {v}
        d(v) = x
    end if
end procedure
```

In fact, a parallel shortest path algorithm, Δ-stepping, was proposed for massively parallel computers by Meyer and Sanders [1998] in as early as 1998. The details of Δ-stepping is listed in **Algorithm** 4.8. It is a generalization of the Dijkstra algorithm and Bellman-Ford algorithm [Cormen et al. 2001]. A

parameter, Δ, is introduced to decompose nodes into buckets with regard to their path costs. The algorithm starts from the bucket with the lowest cost, and finishes until all buckets become empty. The idea is actually to relax the requirement of Dijkstra algorithm, where only the node with the lowest cost can be processed. In the Δ-stepping algorithm, all nodes in the current cost bucket can be manipulated in parallel. Baggio [2007] developed the first GPU implementation for Δ-stepping, but the performance on a G80 GPU is lower than a sequential CPU implementation of the Dijkstra algorithm. The reasons leading to the inefficiency include the synchronization cost for bucket insertion and irregularity of memory accesses. Since later GPUs have improved performance in these aspects, it is appealing to test the Δ-stepping on newer GPUs.

Garland [2008] introduced a method to compute the shortest path using the Sparse Matrix Vector Product (SMVP) kernel. This work is based on the isomorphic structures between the Bellman-Ford algorithm and the SMVP in CSR format. In fact, the SMVP computations involve a series of "+" and "×" operations organized in a certain flow. When "+" and "×" are replaced with "min()" and "+" operators, respectively, the SMVP is then tweaked into the Bellman-Ford based solution of SSSP. The resultant GPU implementation proves to have performance advantages over the respective CPU implementations.

Bleiweiss [2008] proposed an efficient GPU solution for the path-finding problem. The work is based on a data-parallel A* search algorithm [Russell and Norvig 2003]. It attains a decent speed-up in simultaneously finding paths for multiple agents representing virtual creatures in a computer game [Bleiweiss 2009]. One thread is allocated to each agent and independently performs A* search. Note such a set-up is different from the IC routing problem in which certain routing resources are consumed when a wire is routed. The major contribution of this work is the demonstration of a GPU based efficient implementation of the priority queue data structure that is traditionally considered as too complex for the GPGPU model. Each priority queue is exclusively used by a single thread.

Experimental results show that the GPU based A* search has a better scalability than a GPU based Dijkstra algorithm SSSP solution.

The all pair shortest path is originated from SSSP, but requires to derive the shortest path for every pair of nodes in a graph. A cost matrix is usually used to store the length of the shortest path every two nodes. Of course, the simplest way to solve APSP with GPU is to run multiple SSSP in parallel, with each node assigned to a unique group (warp or thread block) of threads [Harish and Narayanan 2007; Okuyama et al. 2008]. However, the method can be inefficient for large and/or dense graphs. One way to improve the performance is to exploit the streaming processing mechanism of GPUs to overlap the transfer of graph data structure and the computations.

The Floyd-Warshall algorithm is the classical solution to the APSP problem [Cormen et al. 2001]. Its pseudo code is listed in **Algorithm** 4.9. Clearly, the algorithm is isomorphic to the matrix multiplication procedure except that the "×" and "+" operators are replaced by "min" and "×", respectively. Katz and Kider [2008] developed a GPU solution for APSP through blocked processing of the adjacency matrix on GPU. The work decomposes the path cost matrix into blocks to be processed by GPU in an iterative manner. In each iteration of the outer loop in **Algorithm** 4.9, all blocks in a distance matrix have to be processed. In [Katz and Kider 2008], the blocks are handled in 3 phases as shown in Fig. 4.20 to guarantee the correctness of results. The matrix based solution for APSP is adopted by a few succeeding works (e.g. [Harish et al. 2009; Buluç et al. 2010; Matsumoto et al. 2011]). A key novelty of the work by Buluç et al. [2010] is that the APSP problem is solved through a Gaussian elimination based recursive algorithm. The path cost matrix is recursively partitioned into smaller blocks and then the path cost for each block is computed by a GPU. This work can deliver a speed-up of 35-75X over the work by Harish and Narayanan [2007].

Algorithm 4.9 Floyd-Warshall algorithm for APSP [Cormen et al. 2001]
input: Nodes *V*, edges *E*,edge weights *W*, where *W(i,j)* storing the weight of edge *(i,j)*, i.e. the distance between nodes *i* and *j*
output: Path cost matrix *D*, where D_{ij} records the cost of the shortest path from node *i* to node *j*

```
begin
    initialize D
    for k =1 to n do
        for i =1 to n do
            for j =1 to n do
                D_ij^(k) = min(D_ij^(k-1), D_ik^(k-1)+ D_kj^(k-1))
            end for
        end for
    end for
end
```

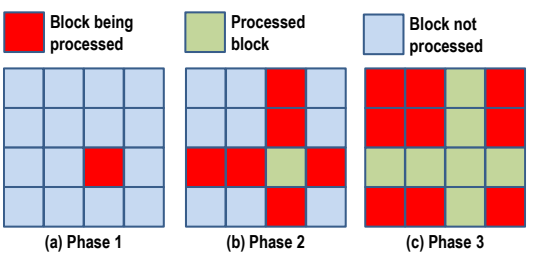


Fig. 4.20 Blocked Floyd-Warshall algorithm [Katz and Kider 2008]

4.3.3 Minimum Spanning Tree

It is a common practice of EDA applications to build minimum spanning trees (MSTs). Especially, it is often used to build initial solutions for various routing problems [Sherwani 1998]. Krushkal's and Prim's algorithms are the most famous MST construction methods [Cormen et al. 2001]. Krushkal's algorithm first sorts the edges according to their weights. It then grows a forest by always picking the least expensive edge that do not leads to a cycle in the current solution. The forest finally merges into a MST. Prim's algorithms starts from a single vertex as the initial solution and also incrementally merges edges in a greedy manner. The bottleneck of Krushkal's algorithm is the sorting process. So it is less efficient than Prim's algorithm on dense graphs. Both algorithms are greedy

and need to choose an edge with the lowest possible weight. Such a nature hinders the available parallelism in Krushkal's and Prim's algorithms. On the other hand, Boruvka's algorithm [Cormen et al. 2001] uses a bottom-up approach to build a MST. At the beginning, every node serves as an individual components. Then each components grows by merging a locally optimum edge. After merging, a single component is collapsed into a single "super-node". This process repeats until a MST is constructed. Obviously, Boruvka's algorithm is more amenable to parallel computing. As a result, all GPU accelerated MST procedures are based on the Boruvka's algorithm [Harish and Narayanan 2007; Vineet et al. 2009; Rostrup 2013].

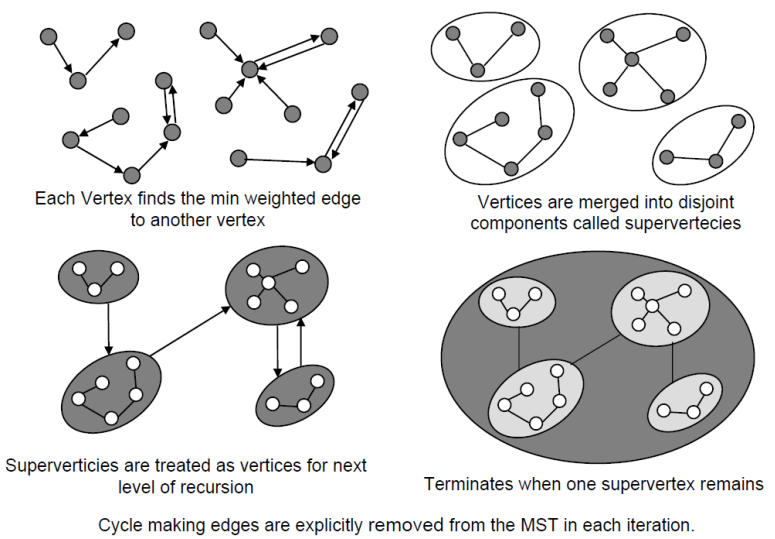


Fig. 4.21 Boruvka algorithm for MST [Vineet et al. 2009]

The GPU based MST procedure developed by Vineet et al. [2009] consists of a series of carefully designed GPU kernels to implement Boruvka's algorithm on GPU. The kernels are organized into a relatively complex algorithmic flow as shown in Fig. 4.21. First, a kernel is started for every vertex to identify the edge with the minimum weight among its outgoing edges. This process can be implemented on GPU as a segmented prefix scan as discussed in sub-section 4.3.1. In the

second step, nodes connected by the minimum-weighted edges located in the first step are clustered into super-nodes. A new graph consisting of both the new super-nodes and the remaining original nodes is then constructed. Cycles must be removed after merging. The previous steps 1 and 2 are performed repeatedly until only one super-node remains. The resultant performance is rather impressive. The GPU implementation outperformed the MST routings (based on Krushkal's and Prim's algorithms) in the Boost C++ graph library [Siek 2002] by around 50 times on random graphs.

The Filter-Kruskal algorithm [Osipov et al. 2009] was proposed to mitigate the costly sorting process of Kruskal's algorithm. It concurrently filters away the heavy edges that cannot be kept in a MST. Rostrup et al. [2013] studied the difference between the Filter-Kruskal and Boruvka's algorithms and then proposed Data-Parallel Kruskal's algorithm for GPUs. At the initialization stage, a high-level partitioning by edge weight is performed to cut the graph into sub-graphs. MST for each sub-graph is then constructed with Boruvka's algorithm. The resultant GPU implementation is very efficient. It outperforms the previous work by Vineet et al. [2009] by up to 10 times on random and real-world graphs. It also delivers a higher level of performance than an 8-core implementation of the Filter-Kruskal algorithm.

4.3.4 Graph Cut and Mincut/Maxflow

The graph cut problem, also designated as the graph partition problem, is to decompose an edge-weighted graph into two or more parts such that the sum of weights of all edges connecting different parts is minimized. It is a popular tool of EDA applications. Note that a circuit netlist usually corresponds to a hypergraph, but can be converted to a graph [Sherwani 1998]. A classical algorithm to solve the graph cut problem is through the mincut/maxflow theorem. If we introduce two artificial nodes, *source* and *terminal*, to a given graph, a maximum flow that can be pushed from *source* to *terminal*, is equal to finding a minimum cut (*source* and *terminal* in different parts, i.e., s/t

cut) of the graph. Please refer to related literature (e.g., [Cormen et al. 2001] for details of the proof.

Algorithm 4.10 Parallel Push-Relabel Algorithm [He and Hong 2010]

```
input: Nodes V including source node s and terminal node t, edges E, edge capacity c_uv for each edge
output: maxflow f
begin
    initialize height function h(u) for each node u, excessive flow e(u) for each node u, residual capacity c_f(u, v)for each edge (u,v), total excessive flow ExcessTotal
    copy e and c_f from CPU main memory to GPU global memory
    while e(s) + e(t)<ExcessTotal do
        copy h from CPU main memory to GPU global memory
        for each vertex in parallel do
            push_relabel_gpu()
        end for each
        copy c_f, hand efrom GPU global memory to CPU memory
        call global_relabel_cpu()
    end while
end
```

Algorithm 4.11 Initialization of Push-Relabel Algorithm [He and Hong 2010]

```
Initialization()
begin
    h(s)= |V|
    e(s) =0
    for all node u∈V − {s}do
        h(u)= 0
        e(u)=0
    end for
    for all (u, v)∈Edo
        c_f(u, v) =c_uv
        c_f(v, u) =c_vu
    end for
    for all edge(s, u) ∈Edo
        c_f(s, u) =c_f(s, u)− c_su
        c_f(u, s)=c_f(u, s) + c_su
        e(u)=c_su
        ExcessTotal=ExcessTotal+ csu
    end for
end
```

Algorithm 4.12 The **push_relabel_gpu** function for vertex *u* [He and Hong 2010]

```
push_relabel_gpu()
begin
    cycle = KERNEL CYCLES
    while cycle >0 do
        if e(u)>0 and h(u) < |V |then
            e' >e(u)
            h' =∞
            for all (u, v) ∈Ef do
                h'' =h(v)
                if h'' < h'then
                    v' =v
                    h' =h''
                end if
            end for
            if h(u) > h'then
                d =min(e', cf(u, v'))
                atomicAdd(cf(v', u), d)
                atomicSub(cf(u, v'), d)
                atomicAdd(e(v'), d)
                atomicSub(e(u), d)
            else
                h(u) =h' + 1
            end if
        end if
        cycle =cycle − 1
    end while
end
```

The mincut/maxflow problem can be solved with two different lines of techniques, augmenting path and push-re-label. The former operates in an iterative manner. In each iteration, it identifies a path from *source* to *terminal* and pushes as much flow as possible through the path. The latter repeatedly performs a push and a re-label action. Given a node with excessive flow, the push action allocates a certain amount of the flow to its neighbors, if possible. The re-label action works on nodes that cannot be pushed to increase the possibility of further push actions. The push-re-label algorithm is generally preferred for parallel execution because it allows concurrent manipulation of multiple nodes.

Algorithm 4.13 The **global_relabel_cpu** function on CPU [He and Hong 2010]

```
global_relabel_cpu()
begin
    for all (u, v) ∈E do
        if h(u) > h(v) + 1then
            e(u=e(u) − cf(u, v)
            e(v) = e(v) + cf(u, v)
            cf(v, u) =cf(v, u) + cf(u, v)
            cf(u, v) = 0
        end if
    end for
    backwards BFS from terminal node, t, and assign the height function with
    each vertex's BFS tree level
    if not all the vertices are relabeled then
        for all u ∈V do
            if u is not relabeled and marked then
                mark u
                ExcessTotal = ExcessTotal− e(u)
            end if
        end for
    end if
end
```

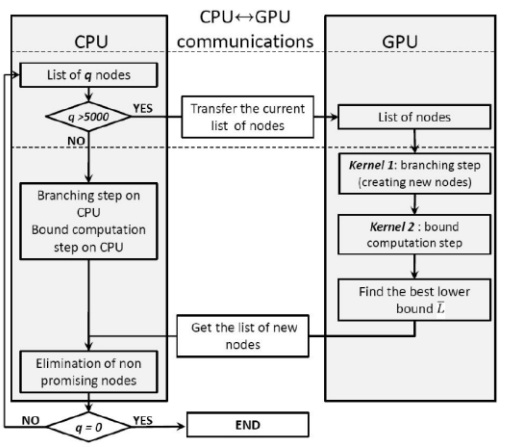
Hussein et al. [2007] and Vineet and Narayanan [2008] proposed works on GPU accelerated graph cut using the Push Re-label algorithm. A more general work was introduced by He and Hong [2010]. The major contribution of this work is a novel CPU-GPU hybrid flow with the top level structure shown in Algorithm 4.10. The corresponding algorithms are listed in Algorithms 4.11, 4.12, and 4.13, respectively. The data-intensive operations of push and re-label are performed by GPU. He and Hong [2010] also developed dynamic tuning techniques for their hybrid flow. The core idea is to allocate workload between CPU and GPU in an adaptive fashion by considering the dynamically available parallelism. The hybrid solution of He and Hong [2010] is capable of processing a 2M-node graph in several seconds. On the other hand, hMetis [Karypis et al. 1997], which is the state of the art graph

partition engine using a sequential algorithm, needs a few minutes to handle a graph with a similar scale[15].

4.4 *Backtrack and Branch-and-Bound*

Backtrack and branch-and-bound (B&B) algorithms are generic search methods to solve combinatorial problems [Cormen et al. 2001]. Both algorithms enumerate potential (partial) solutions and then perform evaluations on them. A current (partial) solution is discarded as soon as it is determined to be either invalid or destined to be suboptimal. Both algorithms have an exponential complexity in the worst case and thus acceleration execution is often necessary.

The backtrack algorithm works in a depth-first manner. The D-algorithm developed for automatic test pattern generation (ATPG) is a typical backtrack algorithm [Wang et al. 2006]. It is also widely used in binary decision diagram (BDD) based formal verification applications [Bryant 1986]. Since the depth-first search usually involves recursive function calls, which are not supported by current GPUs yet, few works address the problem of implementing the backtrack algorithm on GPUs.



[15] It is appealing to quantitatively compare the best approaches of CPU and GPU based graph partition algorithms. To the best of the authors' knowledge, such a detailed comparison is not available yet.

Fig. 4.22 Adaptive hybrid implementation of branch-and-bound [Boukedjar et al. 2012]

The branch-and-bound algorithm allows solutions to be enumerated in a breath-first manner. It requires the upper and/or lower bounds of a partial solution can be accurately estimated. Many logic synthesis and optimization algorithms use the B&B algorithm to identify optimal solutions. In addition, it is also an essential tool to solve the integer programming problem, which has applications in routing and many logic/physical optimization algorithms. GPU accelerated solutions for the B&B algorithm have been studied in a few recent works (e.g., [Boukedjar et al. 2012; Melab et al. 2012]). Due to the irregularity inherent in the B&B algorithm, these works adopted a hybrid approach to allocate workload between CPU and GPU for optimized overall performance.

Boukedjar et al. [2012] proposed a CPU-GPU hybrid implementation of the branch-and-bound algorithm for the knapsack problem. The idea is to concurrently create many new solutions (i.e., branch) and prune out unpromising solutions (i.e., bound). During the optimization process, a list is maintained to book-keep the states (or nodes) that can be branched to generate new solutions. Since the number of states varies across the optimization process and too small a number leads to insufficient usage of GPU resources, a threshold is used to choose between CPU and GPU to perform the branch operation in an adaptive manner. When the number of states is lower than the threshold, a branching operation is performed on CPU. Otherwise, GPU is chosen. The overall flow is shown in Fig. 4.22. The proposed approach has been tested on knapsack problems with problem size of up to 500 and the speed-up over an optimized CPU solution is close to 10X. The performance bottleneck is the atomic operations and the divergent behaviors among threads due to the computation of bounds.

Melab et al. [2012] developed another hybrid implementation for the branch-and-bound algorithm with application in the flow-shop scheduling problem. It differs from the work of Boukedjar et al. [2012] in workload assignment between CPU

and GPU. As demonstrated in Fig. 4.23, the branching operation is exclusively performed by the CPU, while the new states created by a batch of branch operations are evaluated by GPU in parallel. Experimental results show that the performance of the resultant GPU implementation is promising.

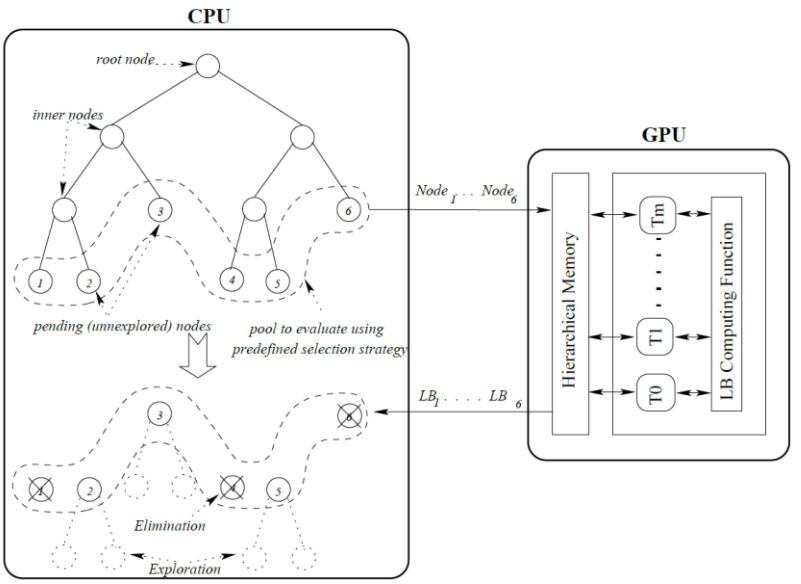


Fig. 4.23 Another hybrid implementation of branch-and-bound [Melab et al. 2012]

Code 4.13 Sample conditional statements ([Chakroun et al. 2012])

```
if (x != 0)
    a = b[1];
else
    a = c[1];
```

Code 4.14**Code** 4.13 refactored by removing conditional statements ([Chakroun et al. 2012])

```
intcoeff = __cosf (x) //hardware supported fast cosine function
a = (1- coeff) *b[1] + coeff*c[1];
```

A succeeding work of GPU based branch-and-bound mitigates the overhead of control divergence [Chakroun et al. 2012]. Generally, the code for bounding operation involves a large number of conditional statements to estimate the lowest and/or the highest cost of a given solution. The conditional instructions incur a lot of performance overhead for SIMD

execution. Chakroun et al. [2012] proposed a few novel solutions. First, a data reordering is performed before data are sent to GPU for bounding operations. The new states are sorted by their positions in the branching tree so as to increase the similarity of data assigned to a single warp. Second, a refactor process can be used to convert certain conditional statement into unconditional. For instance, **Code** 4.13 can be rewritten as **Code** 4.14 without changing any functionality. Here the key idea is to use a cosine function and enforced data type conversion to replace the condition. Modern GPUs have dedicated hardware in the special functional units to compute trigonometric functions. Hence, it only takes two cycles to derive the condition value.

4.5 MapReduce

MapReduce is a parallel programming model to handle large data sets [Dean and Ghemawat 2004]. It was originally designed for distributed computing environments. Taking a divide-and-conquer approach, it first decomposes a workload into independent parts that can be concurrently processed by many processors and/or computers and then merges the partial solutions together. The above first stage is designated as the "map" operation, and the second as "reduce". MapReduce matches the computation structures of big data analytics and finds wide applications in web based large-scale applications. The original MapReduce framework has already been ported to multi-core and GPU platforms (e.g., [Ranger et al. 2007; He et al. 2008]).

In the terminology defined by Catanzaro et al. [2008], the MapReduce concept is actually a generalization of the original idea. In EDA applications, MapReduce typically suggests a divide-and-conquer computing pattern. One typical usage of the pattern appears in stochastic optimization algorithms such as simulated annealing and evolutionary algorithms. The map stage creates many new solutions (partial or complete) through local search, and the reduce stage picks up the most appropriate one.

The simulated annealing algorithm is perhaps the most popular stochastic search heuristic used in EDA applications. With circuit placement as its most successful application[16], the simulated annealing algorithm is also used in routing and circuit optimization [Sherwani 1998]. Coordinated by a statistically optimal schedule, this algorithm iteratively improves its solution derived by local search. In a sequential implementation, the simulated annealing algorithm repeatedly performs two basic operations, generating a new solution (coined as a move in the corresponding literature [Sherwani 1998]) and evaluating the solution for acceptance or rejection. Accordingly, the simulated annealing algorithm can be parallelized in 2 different ways: 1) generating many new solutions in parallel and pick up the best one; and 2) launching many parallel processes with each process executing a sequential version and then take the best result after a synchronization. Early trials of parallel simulated annealing algorithms were proposed by Chu et al. [1999], Onbaşoğlu et al. [2001] and other researchers. However, a complete theoretic treatment of parallel simulated annealing is still open. Researchers have to be careful to check the convergence behavior of their parallel implementations, especially when the number of parallel threads is massive on GPUs. The genetic algorithm [Banzhaf et al. 1998] and evolutionary algorithms [Ashlock 2006] also have applications in EDA and their parallelization on GPU can be treated in a similar manner. A generic methodology for designing GPU-oriented local search algorithms can be found in [Van Luong et al. 2012].

[16] The simulated annealing algorithm used to be very popular in standard cell placers. Now it is mainly used in FPGA placement engines.

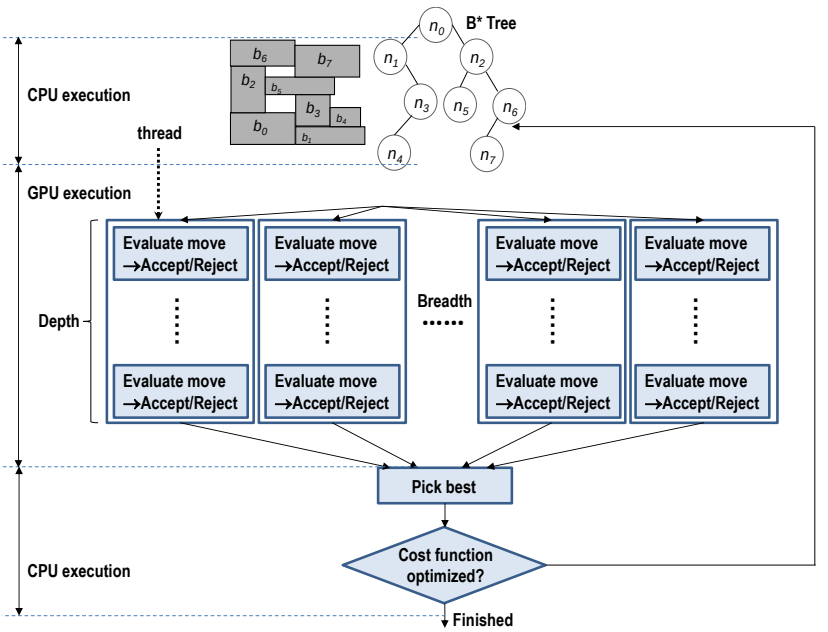


Fig. 4.24 GPU accelerated simulated annealing for floorplan (adapted from [Han et al. 2011b])

Han et al. [2011b] proposed GPU accelerated simulated annealing engine for the floorplan problem. The generic algorithmic flow is demonstrated in Fig. 4.24. The corresponding algorithm is listed in **Algorithm** 4.14. The idea is to launch a GPU kernel for concurrently generating moves, performing evaluations, and then modifying the floorplan representation if necessary, while other processing steps are done by CPU. In general, the operations can be organized in two directions. First, a number of parallel threads can be exploited to perform work concurrently. The number of threads is reflected in a parameter, *Breadth*, in **Algorithm** 4.14. Second, it is also possible to sequentially excise a given number of evaluations by every thread. This number is coined as *Depth*. The sequential evaluations inside a thread are conducted to reduce the chance of being trapped into local minimum. In fact, it is too greedy if a best solution is directly picked after a single move by every threads (i.e., *Depth* = 1). With a proper value of *Depth*, each thread evolves

independently as a canonical simulated annealing process. Then a best solution is chosen only at synchronization points.

Algorithm 4.14 Parallel simulated annealing for floorplan[Han2011]
input: A given circuit with n modules
output: A floorplan that optimizes objective (area, wirelength)
begin
    read input circuit
    construct initial floorplan in a B* tree form
    while stopping criteria not met do
        copy tree and attributes to GPU device memory
        launch *Breadth* parallel thread blocks
        copy tree and attributes to shared memory
        for *i* in 1 to Depth do
            select and perform move
            modify tree /*local copy in shared memory*/
            evaluate objective
            write objective in GPU device memory
        end for
        copy B objectives from GPU device memory to host memory
        pick best move
        modify tree with best move
    end while
end

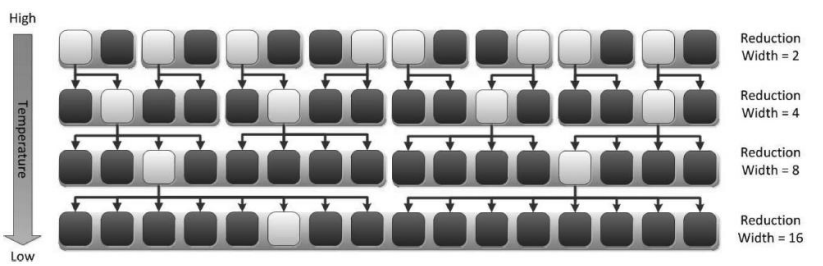


Fig. 4.25 Merging sub-solutions through reduction [Han et al. 2011b]

At a synchronization point during the simulated annealing process, multiple sub-solutions generated by different threads need to be merged to generate locally best solutions. A novel approach to merge sub-solutions (i.e., the reduction in MapReduce) was introduced by Han et al. [2011b]. It is depicted in Fig 4.25. At a higher temperature, a local optimum is picked from every two threads. With the decreasing of the temperature, it is chosen from a larger number of threads. The underlying reasoning is to make the algorithm greedier with

the decreasing of temperature. The merging process can be effectively implemented by the reduction operation of GPU.

Finally, it should be noted that the MapReduce actually has its limitation in support complex applications, where a clear decomposition is hard to find. Accordingly, new parallel programming models targeting the big data analytics applications are emerging. Among these, GraphLab [Low et al. 2010] is worth special attention for EDA researchers because it targets graph based computations.

4.6 Dynamic programming

Dynamic programming (DP) is an efficient method to solve complex problems. A successful application of the dynamic programming method hinges on the assumption that the problem can be decomposed into overlapped sub-problems and the globally optimal solution is the combination of optimal solutions to sub-problems. A classic application of dynamic programming in EDA is technology mapping [Keutzer 1987].

The computing of DP has the same pattern for all applications. Accordingly, we use the Smith-Waterman (SW) algorithm, which is a fundamental tool for DNA sequence alignment [Waterman 1995], as an example to review how to efficiently implement DP on GPU. A large number of works have been dedicated to using GPU to accelerate the Smith-Waterman algorithm for DNA sequence alignment (e.g., [Liu et al. 2006; Khajeh-Saeed et al. 2010]). A Highly tuned open-source GPU implementation, CUDASW++, is publically available [Liu et al. 2010].

$$H(i,0) = 0, 0 \le i \le m$$
$$H(0,j) = 0, 0 \le i \le n$$
$$H(i,j) = \begin{cases} 0 \\ H(i-1,j-1) + w(i,j) & match/mismatch \\ H(i-1,j) + gap(i,-) & deletion \\ H(i,j-1) + gap(-,j) & insertion \end{cases}$$

where m=*length(a)*, n = *length(b)*, $w(i,j) = \begin{cases} match\ score & if\ i = j \\ mismatch\ score & else \end{cases}$

Fig. 4.26 Computation structure of the SW algorithm

The fundamental computation structure of the SW algorithm is listed in Fig. 4.26. The input to SW is 2 sequences or strings, *a* and *b*, each consisting of letters from a certain alphabet. The length of *a* and *b* are *m* and *n*, respectively. A score matrix, *H*, is constructed to find the best alignment of *a* and *b* (i.e., a given position in one sequence from which the remaining sub-sequence is maximally similar to the other sequence). An entry *H(i,j)* reflects the similarity at *a[i]* and *b[j]*. Deriving the values of matrix is performed in a recursive manner as shown in Fig. 4.26. Basically, if *a[i]* matches *b[j]*, a positively score is marked to *H(i,j)*, and vice versa. Note that it is allowed for *a[i]* to match an empty letter, which means a letter has to be inserted to *b*. A gap penalty function is defined for such a purpose. Clearly, *H(i,j)* depends on 3 preceding entries, *H(i-1,j-1)*, *H(i-1,j)*, and *H(i,j-1)*. The dependency of SW algorithms is illustrated in Fig. 4.27. The GPU implementation has to respect the dependencies. All existing implementations thus exploit multiple threads to concurrently compute anti-diagonals of the score matrix, while each anti-diagonal is processed from the upper-left corner to the bottom-right.

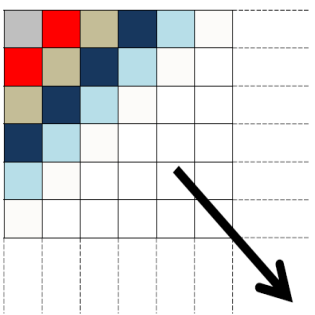


Fig. 4.27 Dependency of the SW algorithm [Xiao et al. 2009]

Xiao et al. [2009] investigate the problem of optimal mapping the SW dynamic programming algorithm to GPU. The essential of the proposed mapping is based on a decomposition of the score matrix into tiles, i.e., the sub-matrices in Fig. 4.28. The tiles along a single anti-diagonal are processed by one call to the kernel, but by different thread blocks. A thread block handles a 2-dimensional tile to improve the efficiency of memory loading. Inside a tile, the processing is still performed

in an anti-diagonal by anti-diagonal manner. Note that the manipulation of a complete anti-diagonal has to be synchronized, although the operations are distributed to multiple thread blocks.

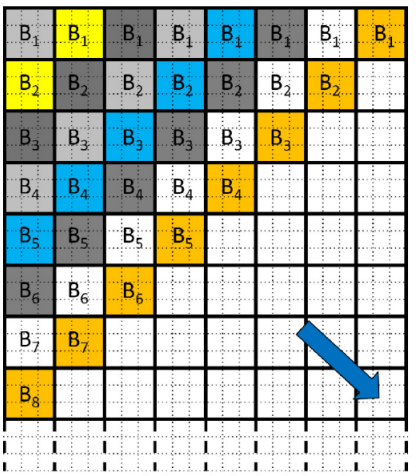


Fig. 4.28 Optimal mapping of the SW algorithm to GPU [Xiao et al. 2009]

Code 4.15 Mutex based GPU global synchronization [Xiao et al. 2009]

```
__device__ void __GPU_sync ( int goalVal )
{
    int tid = threadIdx .x * blockDim .y+ threadIdx .y;     // thread ID in a
    block

    __threadfence ();            // memory flush to all threads
    // only thread 0 is used forsynchronization
    if ( tid == 0) {
        atomicAdd (( int*)&g_mutex , 1);

        // only when all blocks add 1 tog_mutex , will it be equal to goalVal
        while ( g_mutex != goalVal ) {
          // Do nothing
        }
    }
    __syncthreads ();
}
```

Xiao et al. [2009] developed a new global synchronization technique so that a kernel can coordinate all threads without the costly kernel launching. Such a technique is crucial to dynamic programming, which often exhibits abundant

parallelism but requires fine-grain synchronization. Xiao et al. [2009] proposed a mutex based synchronization technique to solve the problem. All active thread blocks atomically writes to a mutex variable in GPU memory for one and only once. When every blocks finishes such an operation, a global synchronization is achieved. Note the limit of this approach is that all blocks of a kernel have to be allocated onto multiprocessors. Otherwise, a deadlock may happen when an active thread block (i.e., a thread block running on a multiprocessor) waits for an inactive thread block to update the mutex variable. The CUDA code of the synchronization method is listed in **Code** 4.15.

4.7 Structured Grid

Among the commonly used computing patterns for EDA applications, the structured grid pattern is the easiest to be implemented on GPU. Its computation is allocated to a uniformly organized grid. It is not necessarily limited to geometric grids. The underlying data structures actually can be any abstract entities. Such a pattern naturally follows a data-parallel computing model. The grid structure can be mapped to the hierarchical GPGPU programming model in a straightforward manner. Due to the relative simplicity of implementing the structured grid pattern on GPUs, we only use an example to briefly illustrate the works in this directions.

Gulati et al. [2009] applied the structured grid pattern to the device model evaluation process in a SPICE-alike circuit simulator. Circuit equations are assembled in the form of matrix for modified nodal analysis (MNA). Every device in the input netlist determines the parameters in a given part of the MNA matrix. This process is designated as model evaluation. Then a linear or non-linear solver works on the equations to derive the timing behavior of a circuit. Each of the two procedures consumes around one half of the total simulation time according to profiling results. The process of model evaluation obviously falls into the structured grid pattern. Gulati et al. developed a GPU based implementation by porting the BSIM3 code [Liu et al. 1999]. The original code was

partitioned into multiple kernels so that each kernel is simple enough for an efficient GPU implementation. For the matrix assembling process, around ~40X speed-up was achieved.

5. GPU Accelerated EDA Applications

The preceding chapter reviewed the GPU accelerated solutions for essential EDA computing patterns. In this chapter, we survey the progress in applying the computing patterns to solve EDA problems on GPUs.

Modern IC designs, especially System-on-Chips, usually start from the system level. The system level design, designated as Electronic System Level (ESL), has a flow as illustrated in Fig. 5.1. The objective of the system level design is to determine the system algorithms, the configuration of communication and computation microarchitectures, as well as the mapping system behaviors to an optimized microarchitecture. At this stage, the major means of verification is simulation, which can be untimed, loosely timed, or cycle-accurate. Please refer to the rich literature of ESL design for details (e.g., [Jantsch 2004; Lin 2010; Martin et al. 2007]).

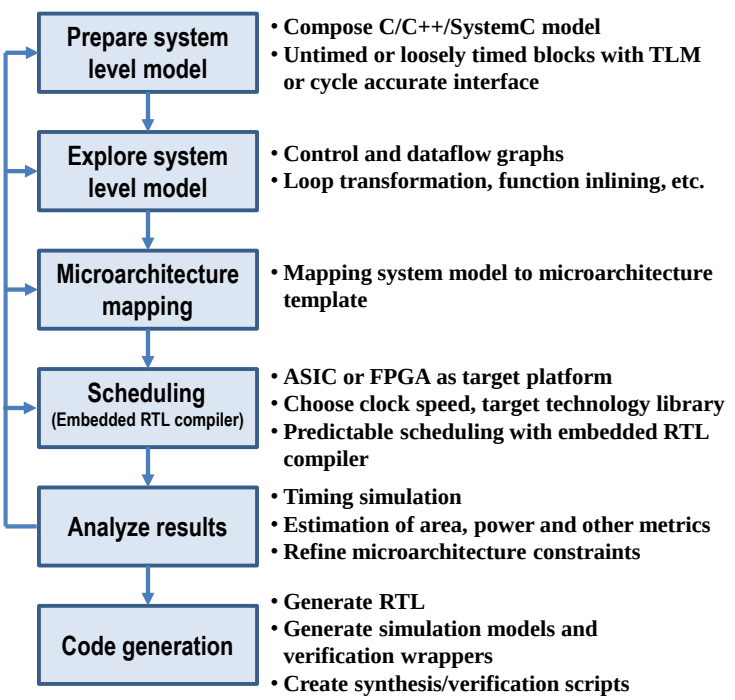


Fig. 5.1 System level design flow (adapted from [Brown 2010]))

After system level design, the next stage for a digital circuit is the register-transfer level (RTL) design. The input is a set of hardware behaviors captured in a hardware description language (HDL). The RTL code can be manually written by IC designers or automatically generated by ESL tools through high level synthesis. The input code will undergo a series of transformations and finally be mapped to a technology library so as to create a gate-level netlist. The succeeding stage is physical design in which the netlist is converted into a manufacturable layout. The functional correctness of digital circuits is generally checked by logic simulation tools, while formal verification tools can be of help. In addition, a large collections of tools are also needed to check the performance, manufacturability, and other physical properties of a design. The design of analog/RF circuits has a different flow. The design is captured by schematics or code in an analog hardware description language. The major verification means is SPICE-alike circuit simulation. The layout of analog/RF circuits is mainly captured manually. We assume the readers are familiar with IC design flows and thus skip the details. Fig. 5.2 is a brief illustration of typical design flows for digital and analog/RF circuits.

In the remaining of this chapter, we review the research works on GPU accelerated EDA at 3 different design stages, system level, RTL, and gate-level. Meanwhile, simulation and other verification means have to be performed across the 3 design stages. Since simulation based verification already becomes the bottleneck of typical IC design processes, considerable research efforts have been dedicated to accelerating various simulation tools with GPUs. So we allocate a separate section to review the works in GPU accelerated simulation.

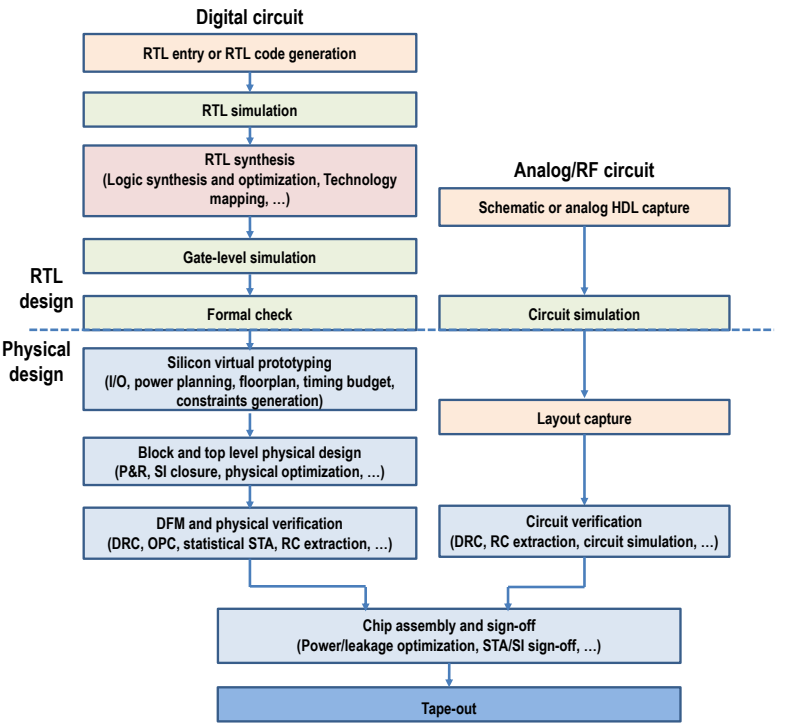


Fig. 5.2 RTL and physical design flow for digital circuits and analog/RF circuits (adapted from [SMIC 2013]))

5.1 System Level Design

The fabrication capacity of today's semiconductor processes allows the integration of a whole system onto a single chip. Given an abstract model that captures complicated system behaviors, ESL tools explore high level design tradeoffs, choose a proper microarchitecture consisting of computation and communication IPs as well as user-defined logic, and schedule and map tasks onto the microarchitecture [Burton and Morawiec 2006]. Another essential design task at system level is high-level synthesis, which automatically converts an algorithmic description in a programming language into synthesizable RTL code.

The system level design process heavily depends on simulation tools to explore a potentially very large solution space. The

simulation of heterogeneous hardware and software system can be extremely time-consuming. Due to the uniqueness of the simulation problem, we postpone the discussion of GPU accelerated system level simulation to sub-section 5.4.1.

5.1.1 Scheduling

Scheduling system tasks and mapping them to a platform are key steps to determine system performance. Complicated analysis is indispensable to find an optimal scheduling and mapping results. A trial-and-error based approach can be tedious and inefficient. As a result, a large body of literature has been established to investigate automatic scheduling and mapping solutions (e.g., [Jantsch 2004; Lin 2010]).

The scheduling problem is long known to be NP-hard. It often requires an excessively long time to compute for real-world problems. Bordoloi and Chakraborty [2010] proposed GPU based solutions for 2 scheduling problems, schedulability analysis and multi-criteria design space exploration.

A task graph consists of a set $T = \{T_1, T_2, \ldots, T_n\}$ of tasks. A task T_i is executed as a sequence of jobs and the 2 succeeding jobs must be separated by an interval of P_i. The worst case execution time of T_i is E_i. The scheduling problem is to find a triggering time for each task such that the deadlines are met and total execution time can be optimized with regard to a certain cost function. Schedulability analysis is a sub-problem of scheduling. It first derives the demand-bound function (DBF) for each task T_i. DBF takes a time interval t as input and captures the maximum execution demand in terms of computing resources. Then the DBFs will be summed together to check if it is feasible to find a schedule for the whole task graph such that the deadlines can be met. Given a set of feasible solutions, the objective of multi-criteria design space exploration problem is to identify the Pareto curve of multiple feasible schedules in terms of implementation cost and performance. Then Pareto-optimal solutions that have the highest performance at each cost values can be derived. An SoC design should adopt a solution from the Pareto-optimal front.

It turns out the 2 problems can both be solved by a dynamic programming technique [Bordoloi and Chakraborty 2010]. A template pseudo code for the two problems is listed in **Code** 5.1. In **Code** 5.1, $c_{i,k}$ is the hardware cost of the kth implementation of task i, while $\delta_{i,k}$ is the expected gain of choosing the kth implementation. $U_{i,j}$ stands for the minimum utilization on a processor that can be attained by only assigned a subset of tasks, $\{T_1, T_2, \ldots, T_i\}$, when the cost is j. An example data dependency relation of **Code** 5.1 is shown in Fig. 5.3.

Code 5.1 Minimum-cost schedulability analysis by dynamic programming [Bordoloi and Chakraborty 2010]

```
kernel () {
    for each pair (δi,k, ci,k) belonging to set Si
        Ui,j= min{Ui-1,j , Ui-1,j-ci,k − δi,k / Pi}
    end for each
}
```

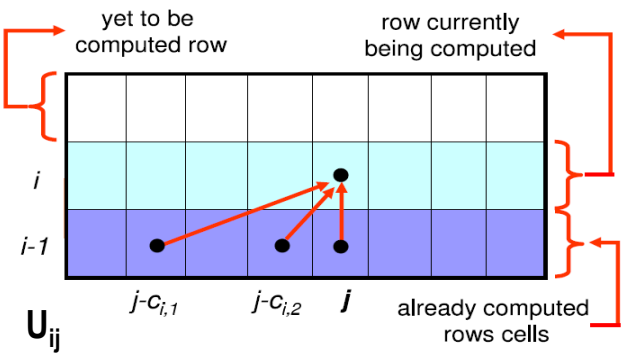


Fig. 5.3 Data dependency in **Code** 5.1 [Bordoloi and Chakraborty 2010]

As a result, the algorithm can be implemented on GPU by following the GPU based dynamic programming pattern introduced in Section 4.6. Bordoloi and Chakraborty [2010] implemented their techniques on a G80 GPU and observed a speed-up of around 10X in both problems. The techniques are promising to deliver a higher level of performance on new GPUs.

Suri et al. [2012] developed an efficient data-parallel solution for the Multiple-Choice Knapsack Problem, which has applications in many system level scheduling and mapping problems. GPU implementations of other scheduling algorithms for large problem instances were introduced by Pinel et al. [2013].

5.1.2 High-Level Synthesis

High-level synthesis (HLS) tools are gradually gaining popularity. Such tools enable a higher design productivity by automatically transforming an abstract behavioral description of a designed system into synthesizable register-transfer level code.

Traditional HLS flows work on a purely abstract algorithmic representation and thus the synthesized RTL design may run into timing closure problems later. Current leading-edge HLS tools incorporate a physical planning engine to guide the high level transformation and optimization (e.g., [Gu et al. 2005]). Williamson et al. [2011] proposed a hybrid optimization framework to optimize supply voltage and control schedule on a unified basis. As depicted in Fig. 5.4, the processing flow starts from a set of initial solutions. Then each solution experiences a series of transformations or optimizations, so-called HLS moves. A move may generate many different candidate solutions. The solutions are sent to GPU to perform a floorplanning and derive a physical prototype. In other words, GPU is used to offload the floorplanning jobs of CPU. The work exploited the computing power of 4 graphics cards to deliver a speed-up of over 1 order of magnitude.

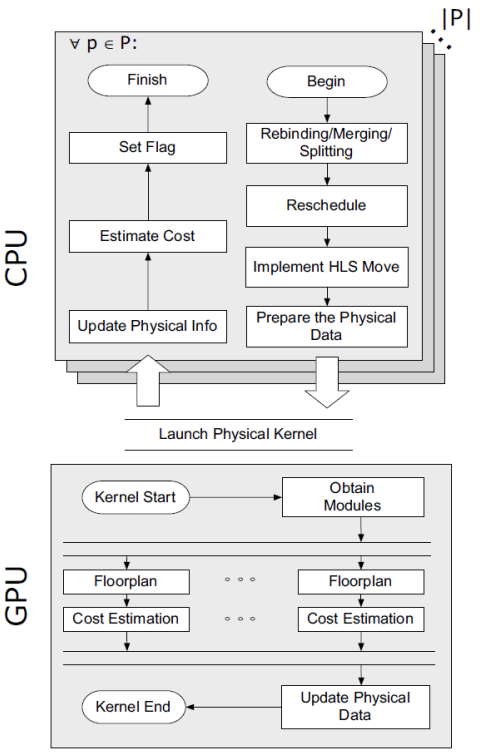


Fig. 5.4 Cross-layer optimization on a hybrid platform (*P* is the set of candidate solution) [Williamson et al. 2011]

5.2 RTL Design

Register transfer level design tools transform an input hardware description into an equivalent logic representation, map it to a technology library and perform logic optimizations. The output of RTL design is a gate-level netlist. Simulation is the most important verification tools for RTL design, while static timing analysis also plays an essential role in evaluating the timing performance of design implementations. We leave the review of GPU accelerated RTL simulation to sub-section 5.4.2 since the problem is a special instance of logic simulation.

5.2.1 RTL Design Implementation

RTL design implementation including synthesis and optimization is typically not the bottleneck of today's IC design flows. However, IC architects still prefer to expedite

this process. Especially, a shorter run time is crucial for FPGA based fast prototyping applications.

Savran and Bakos [2010] developed a GPU based logic minimization procedure. It works on the cube representation of logic. The initial solution is a set of cubes that cover all '1's in the output function. The cubes will undergo a series of operations including conversion, cube absorption, and coordinate subtraction so that they can be transformed into a minimal set of cubes. As depicted in Fig. 5.5, 3 kernels are iterated in a loop. The parallel jobs are straightforwardly organized by assigning a thread to each cube.

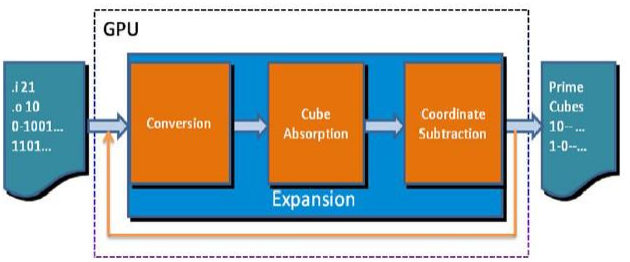


Fig. 5.5 GPU based logic minimization flow [Savran and Bakos 2010]

After library-independent optimizations, a technology mapping algorithm is exploited to map logic elements of a circuit into standard cells in a technology library. A GPU based technology mapper, gpuMap, was introduced by Chen and Singh [2010]. The technology mapping flow is implemented on top of an open-source synthesis tool ABC [Berkeley 2005]. gpuMap consists of a delay-optimal mapping phase and an area-recovery phase. The first phase uses a dynamic programming approach to identify a solution with a minimum number of logic levels from primary inputs to primary outputs. This is done by first levelizing the circuit and then iteratively working on each level. The area-recovery phase aims to minimize the number of k-input lookup tables (LUTs), which is the basic unit of configurable logic, used by the input circuit. This step involves recursive mapping and remapping of logic. Since current GPUs do not directly support recursion, the authors of [Chen and Singh 2010] explicitly store the recursive

jobs into queues. A thread then iteratively fetches a job from the corresponding queue to process. Careful optimizations are critical to extract performance from the GPU implementation. gpuMap is able to generate solutions with a similar quality as ABC does at a throughput that is 3 times higher.

Liu and Hu [2011] proposed a GPU based parallel solution for a circuit optimization problem. It is based on a simultaneous gate sizing and V_t assignment algorithm originally introduced by Liu and Hu [2009]. As listed in **Algorithm** 5.1, the idea is to repeatedly perform topological and reversed topological traversals on a circuit. The traversals follow the spirit of dynamic programming. The objective is to minimize power under timing constraints. During a traversal, different gates tend to undergo identical computations. Such an observation constitutes the foundation of parallelizing **Algorithm** 5.1 on GPU. Each traversal is implemented as a kernel by following the BFS computing pattern. Multiple threads are exploited to assess different implementations of the same gate. Experimental results prove that the GPU implementation achieves the same solution quality as its serial counterpart does, but with a run time shorter by a factor of over 10.

Algorithm 5.1 Joint Relaxation and Restriction (JRR) Algorithm for simultaneous gate sizing and Vt assignment algorithm (adapted from [Liu and Hu 2009; Liu and Hu 2011])

```
begin
    Phase I: Initial solution by relaxation
    history consistency relaxation via topological traversal
    history consistency restoration via reversed topological traversal
    Phase II: Iterative refinement by restriction
    while not converged do
        generate candidate solutions with restrictions via topological order search
        select solution via reverse topological order search
    end while
end
```

5.2.2 RTL Design Verification

Compared with design implementation, RTL verification is facing a much more pressing need for acceleration. The most commonly used RTL level verification tools include RTL simulation, static timing analysis (STA), and formal

verification. We leave the review of RTL simulation in section 5.4.

When a circuit is mapped to a gate-level netlist (not necessarily the final version), it is possible to employ a STA tool to evaluate the timing performance. Unlike logic simulation that needs input vectors to evaluate circuit functionality and performance, STA only uses circuit topology to derive the worst case delay values. STA can be classified into path based and block based algorithms. The former enumerates every timing path and calculates the path delay, while the latter traverses the circuit in a BFS-alike approach. The path based STA has a high accuracy, but suffers from a long run time. Deng et al. [2009] proposed a GPU based method to accelerate the delay calculation for path based STA. The underlying idea is to formulate the delay calculation as a sparse matrix and vector product problem [Ramalingam et al. 2006]. In advanced semiconductor processes, variations of process parameters have been a significant concern that potentially leads to performance failure. Thus statistical static timing analysis (SSTA) tools are required to analyze timing variation under such a circumstance. Gulati and Khatri [2009] proposed a GPU solution for Monte Carlo SSTA. This work levelizes the input circuit and proceeds in a BFS fashion.

Formal techniques are playing an increasingly important role in circuit verification because it proves the correctness in an exhaustive manner. Moreover, there is no need for input vectors. The runtime of formal techniques, however, are always a concern due to the state explosion problem [Clarke and Grumberg 1987]. It is thus appealing to accelerate formal techniques by leveraging the power of GPUs.

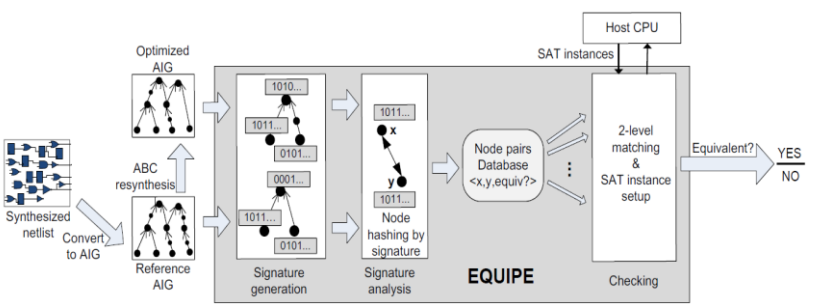


Fig. 5.6 Illustration of the GPU implementation for combinational equivalence checking [Chatterjee and Bertacco 2010]

Chatterjee and Bertacco [2010] proposed a GPU solution to check the equivalence of two combinational netlists. Such tools can be used to check if a circuit after logic transformation and/or optimization still matches the original reference circuit. The processing flow is shown in Fig. 5.6. A key idea is to generate a signature for each node in the netlist through logic simulation. Since the netlists are represented by and-inverter graphs (AIG), efficient GPU based logic simulation can be performed[17]. A hash function is computed on the signature to group nodes into potentially equivalent groups. Then every pair of nodes that comes from different netlists but have the same hash value are checked for equivalence. If two nodes cannot be resolved to be equivalent or not, two AIGs are constructed by considering their fan-ins. Next the two AIGs can be checked by a SAT solver, which runs on the host CPU, to resolve the equivalence. The resultant speed-up of the CPU/GPU hybrid implementation is around 10X when compared with a commercial tool and around 1.5X over the ABC tool [Berkeley 2005]. A key contribution of this work is the demonstration of how a complicated EDA toolkit can be effectively implemented on heterogeneous platforms.

Boolean satisfiability (SAT) solvers are the backbone of many formal verification tools. Despite the importance of SAT in both theoretic computer science and engineering applications, it is still an open problem to develop an efficient GPU solution to Davis–Putnam–Logemann–Loveland (DPLL) based SAT solvers [Biere et al. 2009]. The major hurdles include the highly irregular backtracking process and unbalanced load distribution. An efficient GPU implementations for a survey propagation based SAT solvers was proposed by Qian and Deng [2012]. However, such statistical solvers are incomplete and cannot meet the requirements of many EDA applications.

5.3 Physical Design

[17]A more detailed treatment of GPU accelerated logic simulation can be found in Section 5.4

Physical design is the process of converting a gate level netlist into a manufacturable layout. Physical design involves both implementation and verification tasks. The design implementation tasks include floorplanning, placement, routing, and various optimization steps, while verification tasks cover a wide range of problems from such different domains as timing, power grid, thermal characterization, and manufacturability. Physical design can be extremely time-consuming due to the complexity of today's IC designs and fabrication processes.

5.3.1 Floorplanning

Floorplanning is the starting point of physical design. Originally, floorplanning only concerns the packing of building blocks of an IC design, but modern floorplanners also need to consider such tasks as I/O and power grid planning, timing budgeting, and constraint generation. In this work, we only consider the block packing problem. It can be a laborious process during floorplanning, especially for SoC designs with a large number IPs, although in general it is not the bottleneck in IC design flows.

Han et al. [2011b] developed a GPU accelerated floorplanner. Potential floorplan solutions are represented as B^*-trees and optimized by a simulated annealing engine that runs on a GPU. We already explain the essentials of the GPU based simulated annealing engine in Section 4.5 as an example of the MapReduce pattern. An interesting feature of this work is that the solution quality and run time can be adjusted through adapting the width of parallel execution (number of parallel threads) and the depth of sequential execution (i.e., number of moves per thread).

Williamson et al. [2011] proposed a simplified simulated annealing based floorplanner, which runs on GPU to provide a virtual physical prototype for high level synthesis.

5.3.2 Placement

Circuit placement has been a key step in physical design. Although there exist many placement algorithms, modern commercial placers heavily depend on two algorithms: force

driven analytical placer for standard cell designs and simulated annealing placer for FPGA alike programmable device.

Cong and Zou [2009] developed a GPU based placer. It is based on a multi-level analytical placement framework mPL [Chan et al. 2005]. The analytical placement algorithm takes an analogy from mechanics to solve the placement problem. Cells in a placed netlist are regarded particles in a force filed. A cell is influenced by two forces, an attractive force determined by the interconnection and a repulsive force proportional to the density of cells in the nearby region. Then the minimization of the wire-length can be formulated as a non-linear programming problem. The optimal solution is achieved when the cells are at their equilibrium positions. The mPL placer also exploits a multi-level approach. The netlist is first hierarchically coarsened into smaller ones and placement is performed on each level of the hierarchy. Experimental results prove that the bottleneck of the procedure is the calculation of forces, which can be mapped to GPU for parallel execution in a straightforward manner. In fact, there is already a large body of research on using GPU to expedite the computation of forces in the N-body problem (e.g., [Hamada et al. 2009]). Many of the techniques can be borrowed for efficient force calculation. Although the solution of the nonlinear optimization problem is still performed by the CPU, the work reported by Cong and Zou [2009] achieves a speed-up of around 15X.

A GPU accelerated simulated annealing placer was introduced by Choong et al. [2010] for FPGA placement. It is implemented by taking advantage of the MapReduce computing pattern explained in Section 4.5. The overall organization is listed in **Algorithm** 5.2. It follows the generic flow of simulated annealing except that there are multiple parallel jobs performing the move and evaluation operations. A unique feature is that the new move is not directly generated from a global solution. Instead, this work proposed the concept of subset, which is a group of nodes from the input netlist. Choong et al. [2010] proposed a randomized algorithm to generate a set of subsets by considering the logic connectivity and physical distance among nodes. Random perturbations are

conducted on the subsets. The parallel organization at the top level is to assign a parallel job to manipulate a unique subset. **Algorithm** 5.3 lists the algorithmic flow for such a job. Note that a job is not a thread. Instead, a job has its internal tasks including generating moves and computing the total change of cost. Such tasks are assigned to different threads for data-parallel execution. We mark the data-parallelism on the right of the statements **Algorithm** 5.3. A single arrow means that the operation is done sequentially.

Algorithm 5.2 Parallel simulated annealing for FPGA placement[Choong et al.2010]

```
input: Netlist N
output: placement P
begin
    P = randomInitialPlacement()
    T = Initial_Temperature
    W = Initial_Window_Size
    while stopping criteria not met do
        for M times do
            {Si} = generateSubsets(W,N, P)
            parallelAnneal({Si},N,P,T,W)
        end for
        T = updateTemperature(T)
        W = updateWindowSize(W)
    end while
end
```

Algorithm 5.3 Data-parallel simulated annealing algorithm of function anneal [Choong et al. 2010]
Input: subsets $\{S_i\}$, Netlist N
output: Placement P
begin
    Read subset data into shared memory
    Compute pre-bounding box for each net in subset
    Generate a pool of valid swapping nodes in parallel
    for M moves do
        Select a swap from pool
        Compute changes in cost function per net
        Sum changes across all nets with a parallel scan
        Decide and possibly commit
    end for
    write data to global memory
end

5.3.3 Routing

Given today's highly complicated designs and processes, routing can be a lengthy process. Accordingly, parallel routing has long been considered as ultimately the only way to continuously improve routing throughput. However, it turns out routing is a hard problem for parallel execution. Although relatively efficient GPU solutions have been found for the Dijkstra and A* pathfinding (please refer to section 4.3 for details), the IC routing problems pose extra challenges to GPU implementations. Unlike the pathfinding problem in computer gaming, the routing resources are consumable and their usage has to be updated. As a result, after routing a net, a router has to update the corresponding usage of routing resources. When multiple nets are routed in parallel, expensive locking and unlocking operations have to be done to avoid race conditions. Such frequent synchronization can easily overweigh the performance advantage gained by parallel execution.

Han et al. [2011a] introduced a global router implemented on a hybrid CPU/GPU platform by carefully addressing the above problems. A complete global routing flow is illustrated in Fig. 5.7, while the work by Han et al. [2011a] addresses the most time-consuming stages, initial routing as well as the rip-up and

re-routing. The parallelism comes from routing multiple 2-pin nets [18] simultaneously. To avoid the costly synchronization based conflict resolution, the router dynamically extracts nets that can be concurrently processed. A dependency graph is constructed by considering the routing tiles covered by each 2-pin net. Two nets are independent if the tiles covered by them do not overlap. A unique feature of this work is a two-level scheduling mechanism. The bottom level scheduling is performed by a dedicated thread that collects routing jobs through dynamically updating the dependency graph. It works concurrently with other routing threads. The top level scheduler allocates jobs between CPU and GPU by trying to keep both busy. One issue that has not been investigated is that the dependency in congested areas can be so high that there are no sufficient independents nets. However, rip-up and re-route nets in such areas is often the most costly part in a routing process. The routing thread uses a BFS based approach to search a routing path. The reason is to avoid the overhead of A* and Dijkstra algorithms, which use dynamic data structures like heap. An illustration of the BFS based routing is demonstrated in Fig. 5.8.

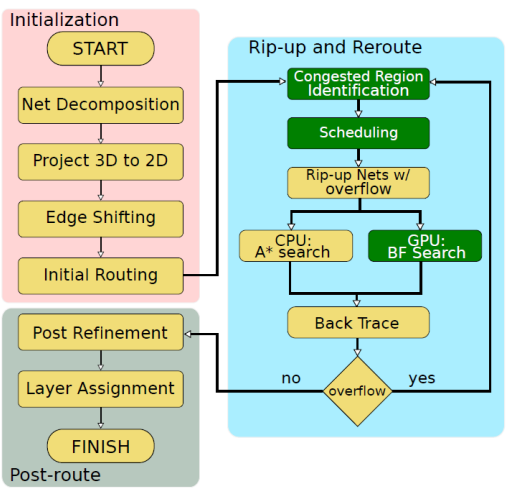


Fig. 5.7 GPU enhanced global routing flow [Han et al. 2011a]

[18]Multi-pin nets are typically decomposed into 2-pin segments in current global routers.

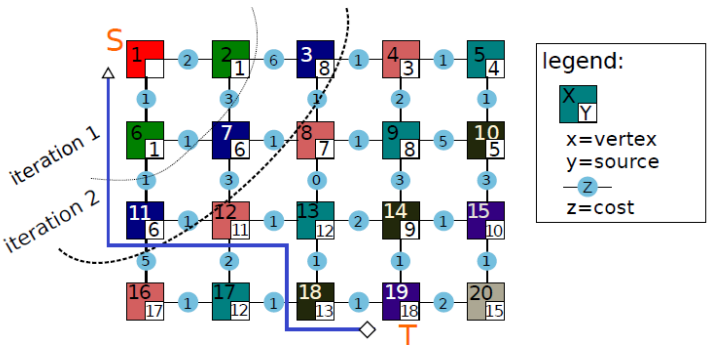


Fig. 5.8 BFS based pathfinding for global routing [Han et al. 2011a]

Experimental results show that the hybrid global router generate high quality routing solutions with a 2-4X speed-up. However, it seems the parallel execution on the 4-core CPU is the main source for the performance enhancement, while the GPU only make relatively minor contributions.

5.3.4 Verification and Analysis

In this sub-section, we review recent research on using GPU to accelerate the verification and analysis of various physical issues in IC design.

5.3.4.1 Capacitance Extraction

Capacitance extraction is an essential task to identify physical parameters from layout. The extracted capacitance information plays a key role in post-layout timing analysis and simulation. Full chip capacitance extraction is a compute-intensive task due to the complexity of modern IC layouts. The fast multi-pole method (FMM), which was developed for the N-Body problem [Greengard and Rokhlin 1987], has been become major technique to solve the capacitance extraction problem and adopted by the well-known FastCap algorithm [Nabors and White 1991]. A few recent works (e.g., [Gumerov and Duraiswami 2008; Hamada et al. 2009]) already developed GPU based FMM solutions, but they cannot be directly in capacitance extraction. Zhao and Feng [2011] introduced processing flows and data structures that can be efficiently integrated in an EDA tool-chain.

Algorithm 5.4 GPU accelerated processing flow for FMM [Zhao and Feng 2011]

Input: coefficient matrices
output: computed capacitance matrix
begin
  Pack all the coefficient matrices for the evaluation cube i (receiver) into a single matrix Φ^i for $i = 1$ *to* n.
  Order the cubes i $(i = 1, ...,n)$ based on the column dimensions of the coefficient matrices Φ^i.
  Cluster Φ^i $(i = 1, ...,n)$ into several new coefficient matrices and generate corresponding index matrices.
  Transfer the coefficient matrices and index matrices to GPU's global memory.
  Characterize the optimal numbers of SMs for workload balancing by running a few small test cases.
  Start the following FMMGpu computation and GMRES iterations:
  for k *in* 0 *to* K do
    Transfer the panel charge vector q to GPU memory.
    Execute the Preconditioning pass on GPU and store the updated charges into Vec_{global}.
    Execute the Direct pass on GPU and store the direct potential contributions into vector p.
    Execute the Upward pass on GPU and store the multipole expansions into Vec_{global}..
    Execute Downward pass on GPU with workload balancing and concurrent kernel executions. Then store the local expansions into Vec_{global}..
    Execute Evaluation pass on GPU with workload balancing and concurrent kernel executions. Then sum the potentials on the evaluation panels to p.
    Transfer the potential vector p back to CPU for the k-*th* GMRES iteration.
    Compute residual $r_o^{(k)}$ after this GMRES iteration, and check the convergence:
    if $r_o^{(k)} < threshold$ then
      Exit the loop and compute the capacitance matrix values.
    end if
  end for
end

FMM consists of 5 major step, all based on the dense linear algebra computing patterns. The corresponding GPU based processing flow is listed in **Algorithm** 5.4. A fundamental issues is that FMM typically uses very small coefficient matrices, which cannot fully exploit the computing power of GPU when computed individually. Zhao and Feng [2011] introduced a method to merge matrices into large ones. A dynamic profiling based technique is proposed to characterize the number of multiprocessors and computing throughput. The

characteristics are used to guide the allocation of multiprocessors when concurrent kernels are launched to compute different set of cube clusters.

5.3.4.2 Thermal Analysis

With the shrinking of feature size and the growing of integration complexity, heat dissipation is becoming a critical concern for IC designs. The problem is even more serious for 3-D ICs in which transistors in top layers are farther away from the heat sinks. To shorten the time for full-chip thermal analysis, a few works have been devoted to GPU based solutions (e.g., [Feng and Li 2010; Liu et al. 2012; Vincenzi et al. 2011]).

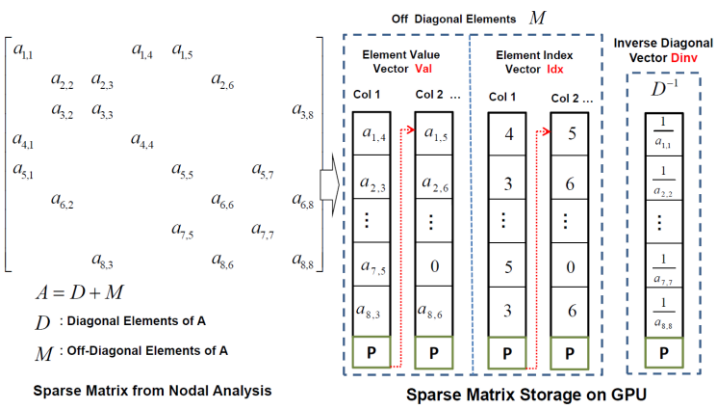


Fig. 5.9 Compact format for thermal conductance matrix [Feng and Li 2010]

Feng and Li [2010] proposed a GPU based full-chip thermal simulation framework. Thermal analysis is achieved by discretizing the thermal PDEs with a finite difference (FD) method. The problem can be formulated as $GT=b$, where G is the thermal conductance matrix, T is the temperature vector, and vector b captures the power sources and boundary conditions. The thermal conductance matrix is sparse and every entry has at most 6 neighbors. As illustrated in Fig. 5.9, the distribution of the non-zero patterns enables the usage of a 1-D vector based format to record all non-zero elements. The resultant linear algebra operations can be efficiently implemented on GPUs. A preconditioned conjugate gradient

based method is used to solve the thermal analysis matrix. The overall speed-up is impressive: the GPU solution outperforms its respective CPU implementation by up to 55-fold.

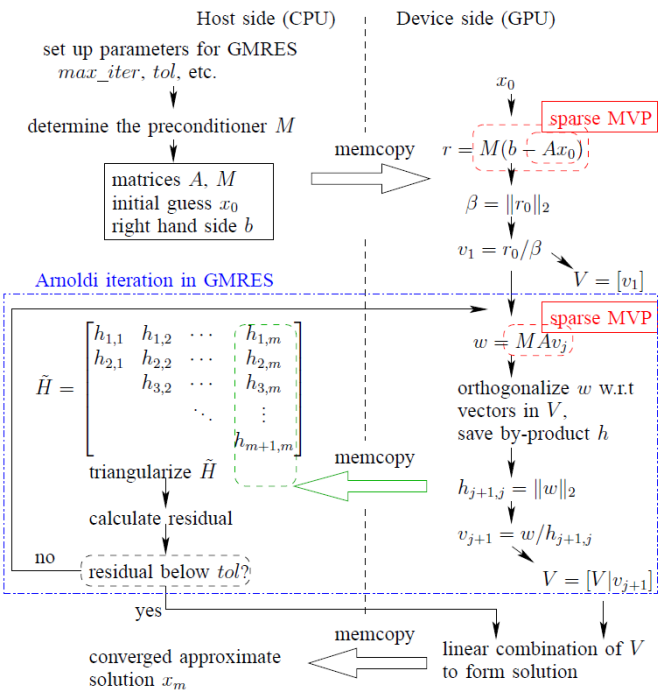


Fig. 5.10 Thermal analysis flow by CPU-GPU cooperation (MVP stands for matrix vector product) [Liu et al. 2012]

Liu et al. [2012] proposed a Krylov subspace based minimum residual (GMRES) method [Saad 2000] based solver for the thermal simulation problem. The compute-intensive matrix and vector operations are implemented on GPU, while the initialization and termination checking are performed by CPU. The thermal analysis is approximated by solving a linear system, $b=Ax$, where x is the temperature vector, A is the thermal conductance matrix capturing the thermal characteristics of a circuit, and b is vector reflecting the impact of heat sources and boundary conditions. Fig. 5.10 shows the flow of thermal analysis. It starts with an initial solution, x_0, and solves the linear system in an iterative manner. M is a preconditioning matrix to improve the convergence speed. The GMRES solver uses Arnoldi iteration [Saad 2000] to generate

the Krylov-subspace with an orthonormal basis as V_m. A new solution is constructed as a linear combination of V_m. The Arnoldi iteration also generates an upper Hessenberg matrix $\widetilde{H}_m$ to check if a solution meets the convergence condition. The resultant GPU solution is 2-4 times faster than a 4-core CPU implementation.

Vincenzi et al. [2011] developed a neural network based simulator to approximate the behavior of thermal system equations. The neural work offers sufficient parallelism to be exploited by GPU.

5.3.4.3 Design for Manufacturability

When feature size of semiconductor process is shrinking below 65nm, Design for Manufacturability (DFM) techniques must be applied to IC layout to guarantee an acceptable level of yield. DFM applications tend to be expensive in terms of CPU time due to overwhelming complicacy of layouts.

Zhang et al. [2009] explored the use of GPU to accelerate the inverse lithography technique (ILT) based mask design. ILT is one of the Resolution Enhancement Technologies (RETs). Considering the design layout as a target, it computes an inverse of the lithography image system function to derive the desired mask. The bottleneck of ILT is a large number of convolution functions, which can be implemented by FFT, which has highly efficient GPU libraries. A 10X speed-up was attained over an equivalent CPU implementation. Similarly, GPU is also used to accelerate the lithography simulation problem [Subramany 2011].

5.3.4.4 Power Grid Analysis

Modern ICs are complicated machines working in extensively varying environments. Robust power delivery systems are essential to guarantee the correct functioning of IC chips. Meanwhile, the complexity of power networks is overwhelming, because they are deployed virtually everywhere on chips and millions of nodes need to be analyzed. Considerable research efforts have been attracted to develop accurate and efficient tools for power grid analysis. The

problem of power grid analysis is to derive DC and transient solutions to a system of linear equations, $GV = I$, where G is the admittance matrix, V is the voltage vector, and I is the vector of drawn currents.

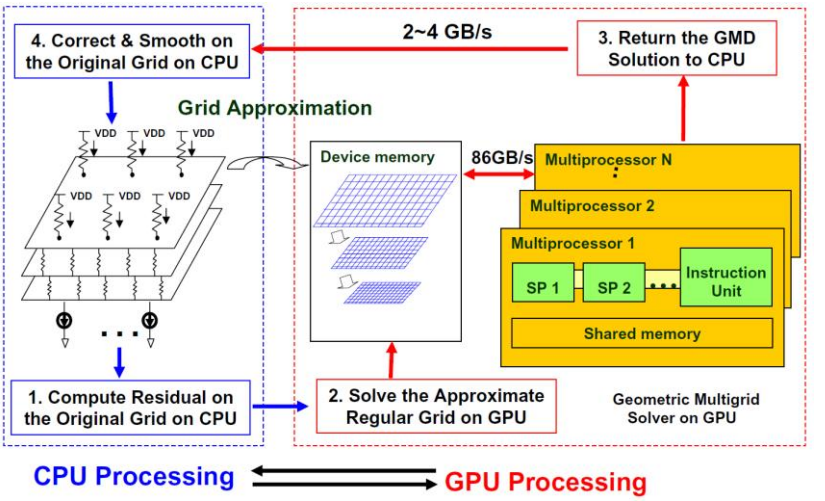


Fig. 5.11 CPU-GPU hybrid power grid analysis flow [Feng and Li 2008]

Feng and Li [2008] pioneered in applying GPU to solve power grid analysis. The overall computing flow is illustrated in Fig. 5.11. The 3-D power grid of an IC is generally irregular. In this work, an approximation scheme is proposed to map the original power grid to a 2-D regular one so that the problem can be efficiently addressed on GPU with the structured grid computing pattern. The regular grid is solved by a geometric multi-grid method. The restriction and relaxation cycles are completely done on GPU, while the residual errors of the 3-D to 2-D approximation is solved by CPU. The relaxation process in which solutions to each grid are iteratively refined dominates the execution time. Feng and Li [2008] proposed two relaxation schemes, Gauss-Seidel iteration and local block weighted Jacobi iteration. The former is used to smooth grids computed by threads from different thread blocks, while the latter is for grid derived by threads in the same thread block. The above idea was extended by Feng and Zeng [2010]. A multigrid preconditioning method is established. The speed-up

against the CPU solution can be as high as 50X on large power grid.

Another early work using GPU to expedite the analysis of structured on-chip power grid was proposed by Shi et al. [2009]. This work tackles with regular power grid. The power grid simulation problem is formulated as a special two-dimension Poisson equation, which can be solved analytically through dense matrix and FFT based techniques. Accordingly, the authors resort to assembling CUBLAS and CUFFT library functions to solve the Poisson equation. By taking advantage the new problem formulation and computing power of GPU, the solution proves to be very efficient and outperform a leading direct solver by a factor of over 10.

Topaloglu [2011] developed a GPU based IR drop analysis solution under an industry setup. The power grid equation, $GV = I$, is solved by a conjugate gradient method. Since matrix G is sparse, the GPU based sparse linear algebra pattern can be applied to this problem. A noticeable feature of this work is the usage of high level GPU programming language, Brook+ [Buck et al. 2004]. A GPU based random number generator [Langdon 2008] is called to simulate the effects of process variations so that three sigma corners can be evaluated.

Zeng et al. [2010] extended the usage of GPU to simulate power delivery networks (PDNs) of electronic systems. Such PDNs deploy multiple on-chip low-dropout regulators (LDOs) to provide stable and accurate voltages. According to the proposed approach, a PDN is partitioned into multiple parts and the respective simulation processes are assigned to both CPU and GPU according to the characteristics of simulation workload. As illustrated in Fig. 5.12, the LDOs and off-chip circuitry are allocated to a SPICE solver running on CPU, while the other more regular on-chip power grid is simulated on GPU.

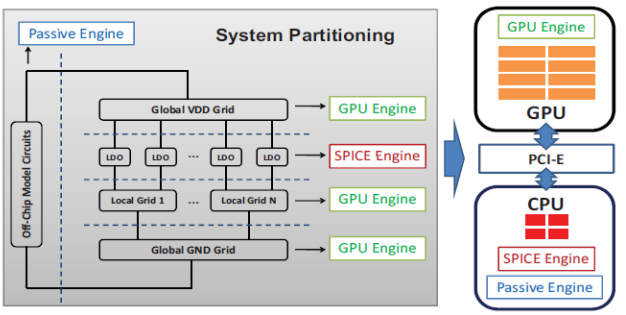


Fig. 5.12 PDN partitioning and mapping to CPU and GPU [Zeng et al. 2010]

5.4 Simulation

The design of IC is to convert an abstract design intention into a manufacturable layout. A complicated flow of synthesis, transformation, and optimization operations have to be performed. After every processing step, IC architects need to verify the correctness of the current design implementation. A wide range of verification tools have been developed. Among these, simulation is still the major means of verification due to its ability to expose arbitrary design details in a manner similar to software debugging. Note that both digital and analog circuits need to be simulated, although different simulation algorithms have to be used.

On the other hand, we are still witnessing a rapid increasing in the VLSI design complexity as the main-stream fabrication process moves to the 32nm and lower technology nodes. For instance, NVIDIA recently released its new generation of GPU, Kepler, which consists of 7 billion transistors [NVIDIA 2012a]. The large scale of VLSI system poses substantial challenges to the verification process. In addition, today's SoC designs are highly complex due to its nature of heterogeneous integration. Both software and hardware in an SoC have to be simulated in a unified framework. As a results, currently the verification process tends to take over 70% of the total design turnaround time in a typical SoC design project [Rashinkar et al. 2000]. For example, the logic simulation of a billion-transistor design can take over one month to finish [Huang 2010]. Another

example is the simulation of phase-locked loop circuits. Given a giga-Herze PLL, it may take a few millions of cycles to track the change of frequency. The simulation of such a complete process will take months on an EDA workstation.

Accordingly, there is a strong need to accelerate the simulation process. In this section, we review the related research progress in 3 aspects, system level simulation, discrete event simulation of digital circuits, and SPICE simulation for analog/RF circuits.

5.4.1 System Level Simulation

Identifying an optimal system architecture and respective algorithms is the ultimate mission of system level design for SoCs. Such an objective is typically realized by a design exploring process in which various options in the solution space are traversed and evaluated. Today the major means to assess the quality of a design solution is still simulation. The system level simulation is inherently a heterogeneous process because it involves hardware (digital and analog) and software captured at different abstraction levels. In addition, the I/O behaviors have to be captured to accurately evaluate system performance under real-world stimuli.

SystemC is the standardized system level design language [IEEE 2011]. It offers a set of C++ compatible language features for capturing system level behaviors. The behaviors of modules and interconnects are encapsulated into processes. The SystemC development team also defines a sequential simulation flow [OSCI 2007]. As depicted in Fig. 5.13, the simulation flow consists of a few phases. In the initialization phase, all processes are executed in an arbitrary order. Then ready processes are sequentially exercised in the evaluation phase and the signal values are refreshed in the update phase. The delayed notification phase and timed notification phase handle events with delay and timing concerns and identify ready processes. Then the simulation transits to the evaluation phase again. Such a loop keeps execution until all processes are running out of events.

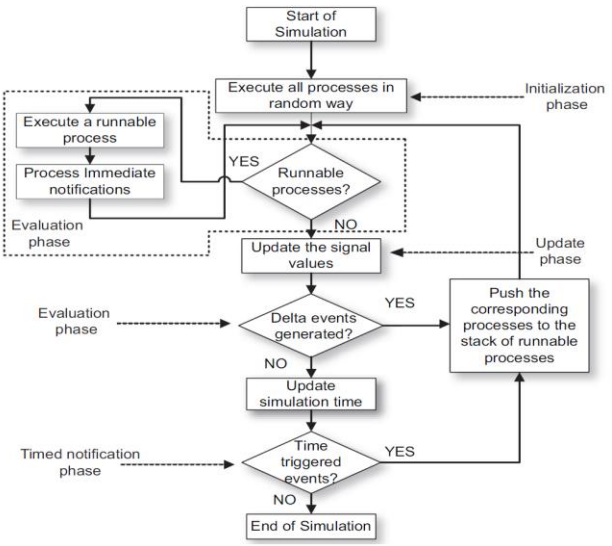


Fig. 5.13 Sequential simulation flow of SystemC [OSCI 2007]

Nanjundappa et al. [2010] proposed a straightforward flow, SCGPSim, to parallelize SystemC simulation. The overall organization of the proposed flow is depicted in Fig. 5.14. A source-to-source framework is developed to translate a synthesizable SystemC description into an executable CUDA program. Since SystemC follows C++ language standard, SystemC processes can be directly mapped to CUDA threads. Since CUDA uses a SIMT programming model and all threads must have the same source code, each thread selectively executes its code if its thread index matches a respective SystemC process. A skeleton of the translated code with many "if" conditional statements is demonstrated in **Code** 5.2. The parallel execution is illustrated in Fig. 5.14.

Code 5.2 CUDA code skeleton for SystemC simulation [Nanjundappa et al. 2010]

```
__global__ static void sample_kernel(arguments) {
    intidx = threadIdx . x ;

    for (i=0;i<MAX_CYCLES; i++) {
            if (idx == 0){
                    // Code that has to be executed by thread 0
            }
            else if (idx == 1){
                    // Code that has to be executed by thread 1
            }
            . . . . . .
            else if (idx == n){
                    // Code that has to be executed by thread n
            }
            __syncthreads ( ) ; //barrier synchronization
    }
}
```

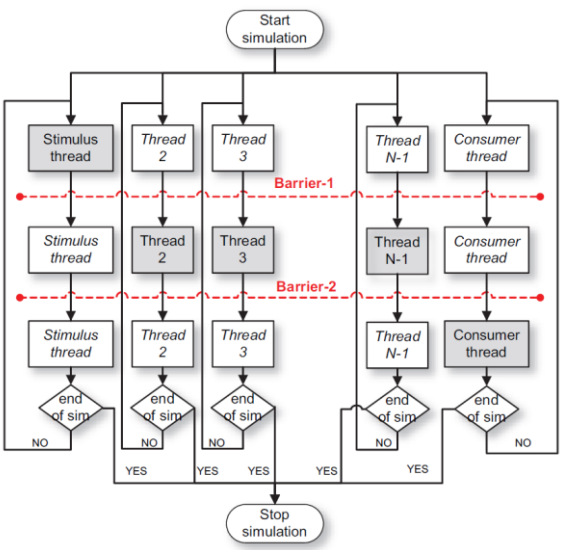


Fig. 5.14 Statically scheduled parallel SystemC simulation on GPU [Nanjundappa et al. 2010]

In the simulation flow proposed by Nanjundappa et al. [2010], threads are statically scheduled. After the GPU kernel is launched, initially only the stimulus thread performs meaningful work to apply stimuli to all other threads. The completion of preparing stimulus is marked by the first barrier. Then evaluation threads start execution in parallel. The

simulation results are written into a buffer. After hitting the second barrier, a consumer thread reads the buffered intermediate results and updates the states of processes. An obvious problem of this work is the explicit introduction of branch divergence into thread code. In addition, the simulation efficiency depends on the inherent parallelism offered by the events occurred simultaneously in the underlying design. The efficiency of such an approach may be problematic on complex designs with sophisticated timing relations.

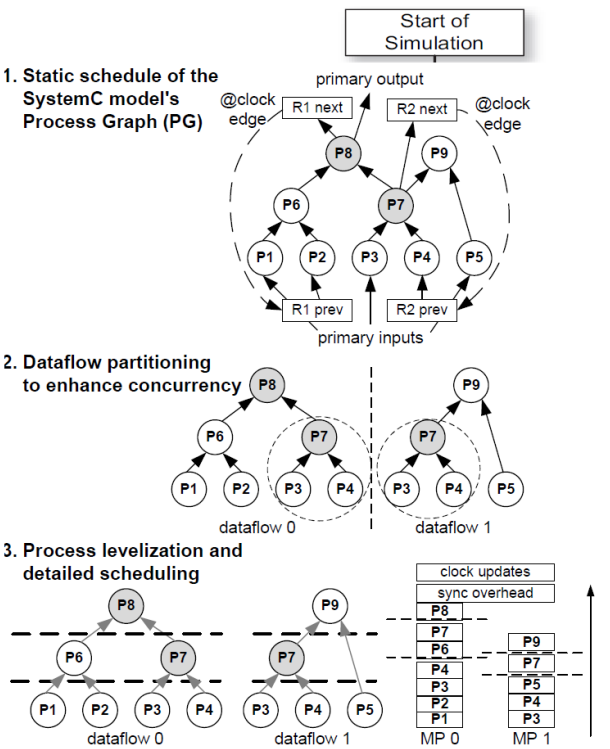


Fig. 5.15 Dynamically scheduled parallel SystemC simulation on GPU [Vinco et al. 2012]

Vinco et al. [2012] also proposed a SystemC simulation framework, SAGA, also by translating SystemC into CUDA code. This work puts an emphasis on SystemC description consisting of RTL modules. It differs from the approach reported by Nanjundappa et al. [2010] in that it adopts a dynamic scheduling strategy. As demonstrated in Fig. 5.15, the idea is to extract a data flow graph (DFG) from the input

SystemC code. A node in the DFG is a SystemC process. The data flow graph is then partitioned into clusters that can be simulated in parallel. In other words, these clusters can be assigned to different warps that executed independently. Next the processes are levelized according to the dependency and simulation is performed in a level-by-level manner. Experimental results shown that SAGA outperforms SCGPSim [Nanjundappa et al. 2010] by factor of 1.7 to 15.

Bombieri et al. [2012b] implemented the abovementioned SAGA simulator with OpenCL and compared it with the CUDA implementation. Results show that the CUDA implementation delivers a rather large performance advantage. The fact can be explained by a few reasons. First, CUDA compilation tools and runtime are more mature. Second, CUDA allows platform-specific optimization, but it is not possible with OpenCL due to its pursuit for cross-platform compatibility.

The above works demonstrate the feasibility of accelerating SystemC simulation with GPUs. However, the existing works only tackle functional simulation of modules made of digital logic. It is still an open problem to handle timed simulation of more heterogeneous SystemC descriptions.

Besides SystemC based simulation, instruction set simulation (ISS) is also an essential tool for system level design to capture the behaviors of the embedded processors. Raghav et al. [2010] and Pinto et al. [2011] proposed a solution to use GPU for accelerating the simulation of thousand-core processors. The idea is to deploy one thread to simulate one core as demonstrated in Fig. 5.16. The thread takes care of the complete flow of instruction execution. Results prove that up to 4096 simple cores can be simulated. Branch divergence is a serious concern of the proposed approach.

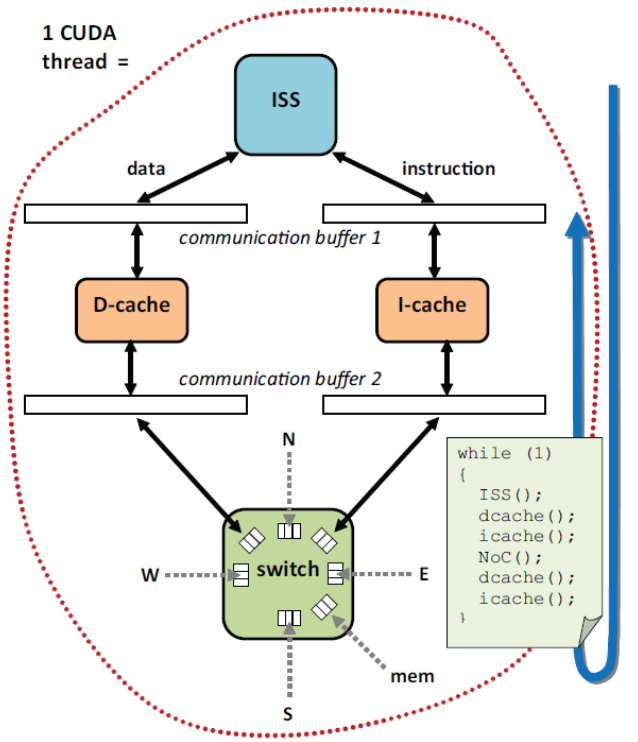


Fig. 5.16 Instruction set simulation by a thread [Pinto et al. 2011]

The above works are extended by Raghav et al. [2012] to offer the capability of full-system simulation. As illustrated in Fig. 5.17, the target system consists both a host ARM CPU and a many-core accelerator. Applications running on the target platform is composed of both sequential and parallel code, which are executed by the CPU and accelerator, respectively. The full-system simulation is performed by both the CPU and GPU of the simulation platform. A QEMU [QEMU] simulator runs on the CPU to emulate the behavior of the host CPU, while the parallel code for the accelerator is translated to CUDA code to be run by GPU.

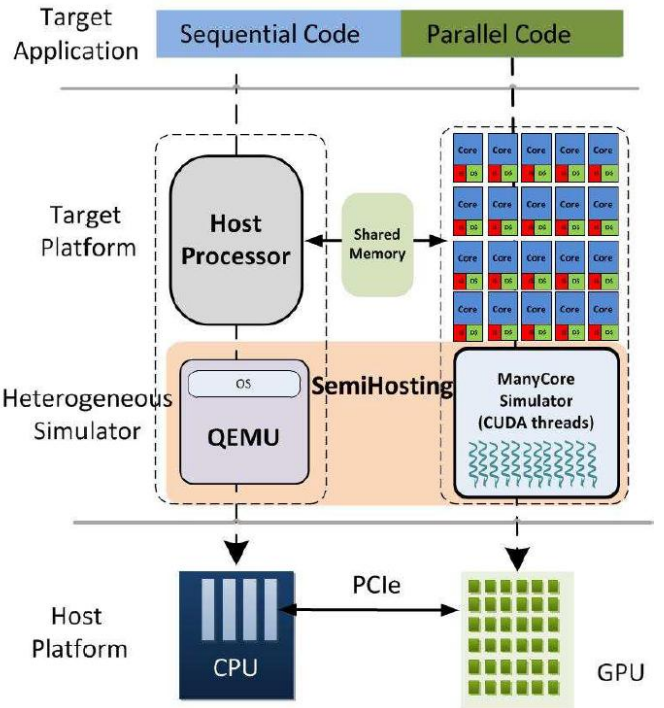


Fig. 5.17 GPU accelerated full-system simulation framework [Raghav et al. 2012]

Despite the above works, there are still many problems to be resolved before GPUs can be used to accelerate more complicated multi-core and many-core designs.

5.4.2 Discrete Event Simulation for Digital Circuits

Logic simulation is perhaps the most time-consuming EDA application in today's IC design flows. The reason is simply because the verification complexity is an exponential function of the design complexity[19]. Although today no IC design teams have the luxury to perform exhaustive simulation, a given simulation coverage still has to be attained to ensure the design quality. Especially, today's IC design projects requires an investment at the scale of $100M and thus a high test coverage

[19]Given a circuit with N bits of state registers and M bits of primary inputs, we need 2^{M+N} test patterns (within the 2-value logic system) to exhaustively verify the correctness of the functioning of the circuit.

is essential for the first-silicon success. Accordingly, it is appealing to leverage the computing power of GPUs for logic simulation.

In this sub-section, we review the research progresses on using GPU to accelerate logic simulation. We start from fault simulation for its relative simplicity, and then review gate-level simulation and RTL simulation. The previous two simulation applications work on gate-level netlists, while the third manipulates RTL designs captured in a hardware description language. Note that the computing pattern of logic simulation, especially timed simulation, is totally dynamic [Pingali et al. 2011] and the patterns introduced in the previous chapter cannot be directly applied.

5.4.2.1 Fault Simulation

As a vital step of the test generation process, fault simulation evaluates how many faults can be identified by a given set of test vectors. Given today's circuit complexity, parallel fault simulation has attracted considerable research efforts. The existing works can be categorized into 3 groups: parallelize the simulation algorithm, partition inputs circuits for concurrent processing, and simultaneously applying multiple test patterns [Gulati and Khatri 2008]. The first 2 approaches are generic techniques that also used in generalized logic simulation problems, while the third one is unique to fault simulation because test patterns tend to be independent with each other[20].

Gulati and Khatri [2008] developed the first GPU accelerated fault simulator. The parallel work has a two-level organization as illustrated in Fig. 5.18. First, the input circuit is levelized and circuit elements at each level can be simulated in parallel. This follows a BFS computing pattern. Fault detection is conducted at the last level. Another level of parallelism comes from the simultaneously evaluations of multiple test patterns. Such a parallel organization delivers relatively good scalability.

[20] For traditional stuck-at fault model, test patterns are generally independent. However, a sequence of dependent patterns may be needed to test a delay fault. Unde such a circumstance, the sequences are independent if a sequence is treated as an entity.

However, the performance of this work still cannot match a carefully optimized CPU based fault simulator [Kochte et al. 2010].

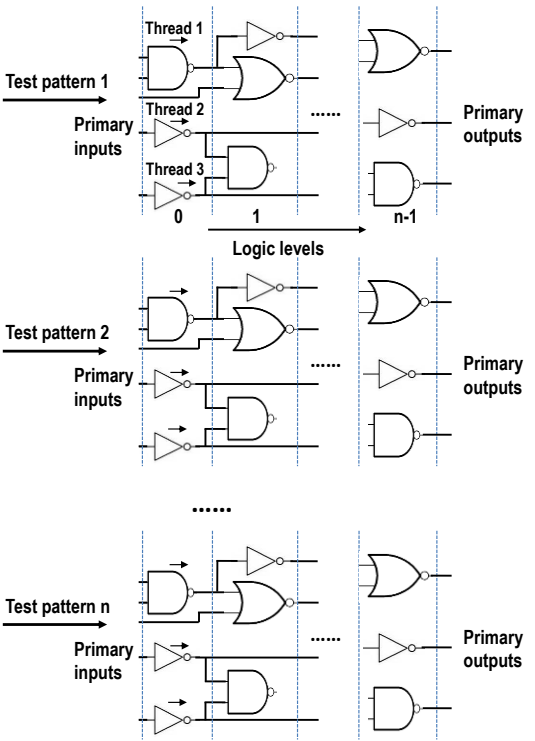


Fig. 5.18 Fault simulation on levelized netlist (adapted from [Gulati and Khatri 2008])

Since threads in warp tend to simulate different logic gates with varying behaviors, the branch divergence is a serious concern. To minimize the divergence, Gulati and Khatri [2008] proposed to use a truth table stored in GPU's constant memory to perform gate evaluation. Different behaviors of logic gates can be derived by the same table lookup operations. This simple technique proves to be quite essential. However, the truth table for industry level design can be very large. If it cannot fit into the constant memory with a limited capacity, the memory divergence due to accessing different memory addresses will also be a performance bottleneck.

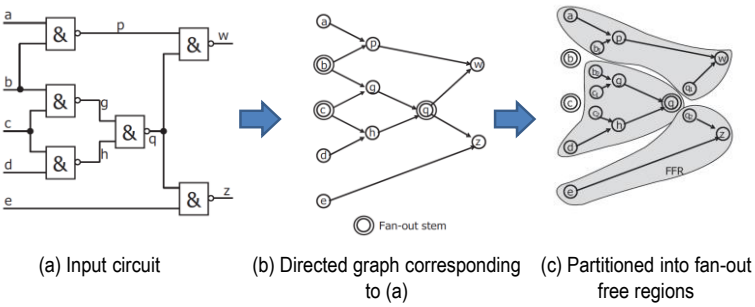


Fig. 5.19 Partitioning input circuit into fan-out free regions (adapted from [Kochte et al. 2010])

Kochte et al. [2010] proposed an enhanced solution to GPU based fault simulation. A directed graph is first constructed according to the input netlist. A key observation is that the fault detection at a node can be done locally if it only has a single successor (i.e., fan-out free) in the graph [Abramovici et al. 1983]. In other words, the fault simulation of such a node can be done independently with others. Accordingly, Kochte et al. [2010] perform a preprocessing on the directed graph by partitioning it into fan-out free regions (FFRs) as depicted in Fig. 5.19. Only the output of a FFR, i.e., fan-out stem, has 2 or more succeeding nodes. Such FFRs are the basic unit of parallel execution. The overall GPU based algorithm is listed in **Algorithm** 5.5. It consists of 3 steps. Steps 1 and 3 are performed in a levelized fashion with nodes in a topological level executed in parallel. FFRs are concurrently processed in Step 2 by starting from fan-out stems. The work delivers superior results over an optimized CPU based fault simulator and the GPGPU simulator proposed by Gulati and Khatri [2008].

Algorithm 5.5 GPU accelerated fault simulation (adapted from [Kochte et al. 2010])
Input: Input circuit as a directed graph *G*, partitioned FFRs, test patterns
output: A list of detected faults
begin
    Step 1: Forward traversal of the graph
    for each vertex *v* in *G* do in parallel
        Evaluate Boolean function associated with *v* for fault-free simulation.
        Compute of the local sensitivities and propagate sensitivities up to the fan-out stems.
    Step 2:
    for each fan-out stem *f* do in parallel
        Evaluate fault propagation if *f* observes sensitized faults
    end for each
    Step 3: Backward traversal of the graph
    for each vertex *v* in *G* do in parallel
        If *v* has only one successor, evaluate the sensitivity of the vertex based upon the sensitivity of the successor.
        From the sensitivity of *v*, determine the observability of the faults and update the fault list.
    end for each
end

A so-called 3-D fault simulation is proposed by Li and Hsiao [2011]. Similar to the approach proposed by Kochte et al. [2010], this work also identifies the inherent parallelism made possible by the existence of fan-out free regions. Basically, faults sharing the same fan-out free region are compacted into a single virtual fault. Such faults are designated as compact faults set (CFS). Different CFS's can be simulated in parallel. Meanwhile, different faults inside a CFS can be evaluated concurrently. In addition, inside each CFS, a gate is simulated by multiple threads for multiple input patterns. As shown in Fig. 5.20, there are 3 different sources of parallelism to be exploited. Experimental results show that this approach attains a higher performance than the work proposed by Gulati and Khatri [2008].

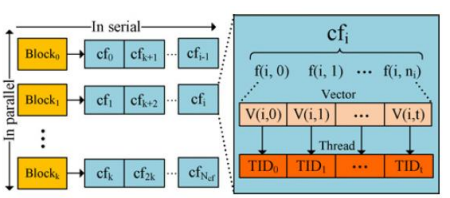


Fig. 5.20. GPU based 3-D fault simulation [Li and Hsiao 2011]

5.4.2.2 Gate-Level Simulation

Gate-level simulation works on a gate-level netlist. Simulation stimuli represented as input vectors are excised on the primary inputs of a netlist so that the output vectors can be validated for correctness. Compared with fault simulation, gate-level logic simulation is a more generalized concept. Generally, the stimulus patterns consist of multiple sequences of test patterns and test vectors inside a sequence are correlated. Such test vectors inside a sequence cannot be treated as independent. The pattern parallelism, which is prevalent in fault simulation, cannot be applied to gate-level simulation.

Gate-level simulation can be either cycle-accurate or timed. Cycle-accurate simulation does not consider exact timing information, but focuses on the purely logic behaviors after clock transitions. Given a topologically sorted logic cone starting from the clock, the evaluation can thus be performed in a level-by-level fashion. On the other hand, timed simulation features a totally dynamic computing pattern. It is infeasible to accurately predict the simulated behaviors without really conducting the simulation.

The event driven algorithm is the most commonly used sequential simulation algorithms for discrete event systems [Fujimoto 2000]. Under the context of logic simulation, a logic transition of a signal pin at a given moment is called an event. The event driven algorithm uses a priority queue to maintain pending events to be processed. At each evaluation cycle, the simulator extracts the earliest event from the top of the queue to evaluate. The algorithm is work-efficient because a signal pin is evaluated only if it has an event. Nevertheless, it is a hard problem to parallelize the event driven simulation algorithm [Bailey 1994]. First, a global event queue has to be maintained. Parallel processes have to synchronize with each other to access such a centralized data structure without leading to race. Second, it is tough to extract parallelism directly from the event queue. Generally, only events happening at exactly the same time can be processed in parallel. Such parallelism is usually inefficient for massively parallel machines. This problem is less serious in cycle-accurate

simulation because all gates are assumed to have a zero-delay. Thus the number of simultaneous events can be large enough.

A counterpart algorithm to event driven simulation is the oblivious algorithm. According to this algorithm, every gate is evaluated at every simulation cycle no matter if there is really an event happening. Apparently, this approach offers much more sufficient parallelism, but involves a lot of useless evaluations. Many parallel algorithms mix event driven and oblivious algorithms for optimized simulation throughput.

Chatterjee et al. [2007; 2009; 2011] pioneered in using GPU to accelerate cycle-accurate gate-level logic simulation. The overall flow is depicted in Fig. 5.21. The input netlist is first levelized and partitioned into macro-gates as illustrated in Fig. 5.22. A macro-gate is formed by starting from a node on the boundary of a level and growing backwards by adding the "cone" of logic. Each macro-gate is mapped to a thread block, which only works when the corresponding sensitivity list of the macro-gate has active events. The logic gates inside a macro-gate is simulated in a level-by-level manner, while gates at the same level can be processed in parallel. A macro-gate balancing operation is also performed on every macro-block to maintain a better load balance by restricting the internal logic. Experimental results show that the GPU based simulator outperforms a commercial sequential simulation by 4 to 44 times. A similar flow is also adopted by Sen et al. [2011].

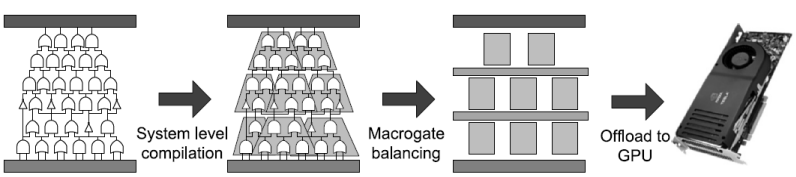


Fig. 5.21 GPU based logic simulation flow [Chatterjee et al. 2011]

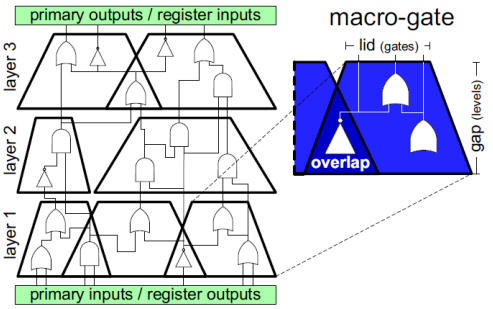


Fig. 5.22 Macro-gate extraction [Chatterjee et al. 2009]

The previously reviewed GPU-accelerated logic simulation algorithm only handles events that happen simultaneously. The parallelism may be insufficient, especially for timed simulation. An alternative approach is the well-known Chandy-Misra-Bryant (CMB) algorithm [Bryant 1977; Chandy and Misra 1979]. According to CMB terminology, different modules of a simulated system are abstracted as logic processes (LP). A LP maintains its local simulation time and have one or more inputs and an output. Different LPs exchange information by sending messages through FIFOs. The key idea of the CMB algorithm is that different LPs are allowed to independently push forward their evaluation as long as the causal relations are not violated. Fig. 5.23 is an illustration of the CMB algorithm. In the 3-gate circuit, the gate *e* has a pending event at 10ns and this fact is passed to gates *f* and *g* as a message. Meanwhile, input pin *a* has an awaiting event at 9ns, while input pin *d* has one at 7ns. Obviously, since *e* will not change until 10ns, the states of *f* and *g* before 10ns are completely determined by *a* and *d*, respectively. Therefore, gates *f* and *g* can be evaluated in parallel because *a* and *d* are independent.

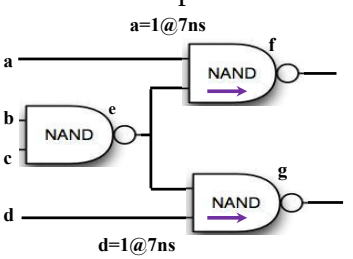


Fig. 5.23 Asynchronous parallelism of CMB algorithm

Wang et al. [2010] pioneered in applying the CMB algorithm to GPU based gate-level simulation. The work was later enhanced by Zhu et al. [2011]. In their works, the global event queue is replaced by distributed queues allocated for each gate. Since it is costly to implement priority queues on GPU, the event queue of a gate is actually realized by multiple FIFOs on its input pins. The earliest event among top entries of each input FIFO is safe to be evaluated. Safe events are processed in parallel by different threads. Note that the size of these FIFOs cannot be statically determined. So a dynamic GPU memory manager is developed to allocate memory at runtime. At the beginning of simulation, the memory manager allocates a large chunk of GPU memory as a pool organized into 4KB pages. The pages are dynamically allocated to FIFOs and recycled to the pool. The manager runs on CPU and exchange information with GPU via the zero-copy mechanism.

The overall algorithm proposed by Wang et al. [2010] is shown in **Algorithm** 5.6. The simulation flow is organized as three consecutively executed kernels, *extract*, *fetch*, and *evaluate*. First, primary inputs extract pending input patterns and insert them into their corresponding event queues. Next, the queues of the pins *fetch* event from all gates (including primary inputs). Finally, in compliance with timing orderings, the real gates (i.e., all gates excluding primary inputs) evaluate the inputs to generate new events and add them into the event queues of the corresponding output pins. The three kernels are executed in a loop until there are no more events or a given number of simulation cycles have been reached. The proposed GPU simulator is faster than a commercial sequential simulator by 3 to 138 times. The simulation flow and available parallelism for each kernel are illustrated in Fig. 5.24.

Algorithm 5.6 GPU accelerated CMB simulation (adapted from [Zhu et al. 2011])

```
Input: Input circuit,  test patterns
output: Simulation results
begin
      while not finished
        // kernel 1: primary input update
        for each primary input (PI) do in parallel
           extract the first message in the PI queue;
           insert the message into the PI output array;
        end for each
        // kernel 2: input pin update
        for each input pin do in parallel
          insert messages from output to input pins;
        end for each
        // kernel 3: gate evaluation
        for each gate do in parallel
           extract the earliest message from its pins;
           evaluate messages and update gate status;
           write the gate output to the output array;
        end for each
      end while
end
```

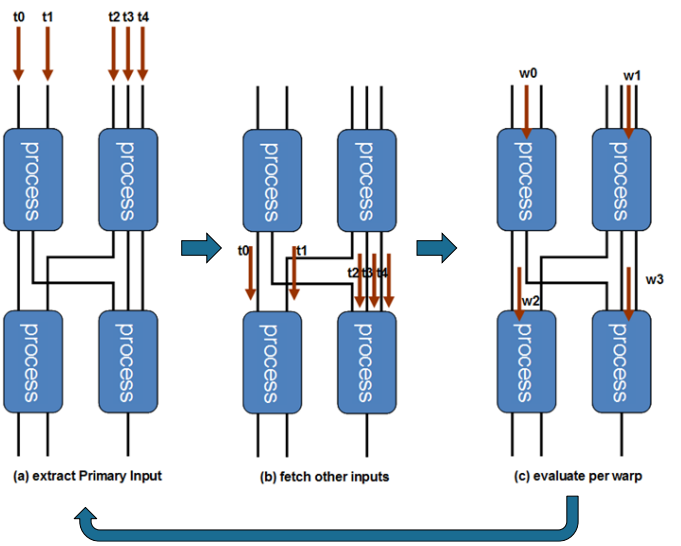


Fig. 5.24 Simulation flow and available parallelism of GPU based CMB simulation [Zhu et al. 2012]

The GPU based logic simulation techniques proposed by Chatterjee et al. [2007; 2009; 2011] and Wang et al. [2010] are complementary. The former is a cycle-accurate simulator and

only considers pure functional behaviors after a clock transition. It focuses on parallelism in the same level of logic. The latter is more general and can simulate circuits with detailed delay information. It offers chance to derive parallelism resulted from the independency of events. Meanwhile, it is also possible to treat each merged gate in [Chatterjee et al. 2007; 2009; 2011] as a unit logic process and use the conservative simulation protocol to extract more parallelism.

5.4.2.3 RTL Simulation

When the algorithm and architecture of an IC design is determined, IC architects capture their design intention in RTL code using a hardware description language (HDL) like Verilog and VHDL. Before synthesized into a gate-level netlist, the RTL code has to be carefully simulated to verify the correctness. The RTL code of leading-edge IC designs integrates complex behaviors. In addition, complicated software behaviors often have to be evaluated on the basis of RTL code. The resultant RTL simulation can be very expensive in run time.

RTL simulation is performed on the HDL code, which poses challenge to GPU based parallel computing. In gate-level simulation, the overall behavior of a design is a combination of a fixed set of standard cells. However, the RTL HDL code may have arbitrary behavior and cannot be decomposed into a fixed set of basic functions.

Qian and Deng [2011] proposed the first GPU based parallel RTL simulation framework. This work takes a compiled simulation approach as illustrated in Fig. 5.26. Verilog code general consists of interactive processes embodied as *initial* and *always* blocks. Such processes can be considered as the basic unit of concurrent execution. In the compiled simulation developed by Qian and Deng [2011], each *initial* and *always* block is translated as a device function running on GPU. The translation process needs to consider the semantic differences between HDL and software programming languages. Meanwhile, the interconnections among processes are also

converted into FIFOs. Every process is mapped to a unique warp. The warps are scheduled by a CMB parallel simulation protocol. Upon activation, a process calls the corresponding device function to simulate the respective behaviors.

The above work provides a working flow for GPU based RTL simulation. It still needs to be enriched in a few directions. First, million-gate level RTL designs will require multi-GPU simulation solutions. Second, a theoretic framework is critical to provide provable solutions to initialize the parallel simulation process without live-locks and avoid dead-locks during simulation. Finally, theoretic analysis is also needed to precisely predict the expected speed-up for a given circuit structure so that IC architects can select a proper simulation platform. Of course, a general solution for this purpose is challenging due to dynamic and irregular nature of the parallel simulation platform. A synergy of computer architecture, compiler technology, and algorithm design will be indispensable to resolve the problems.

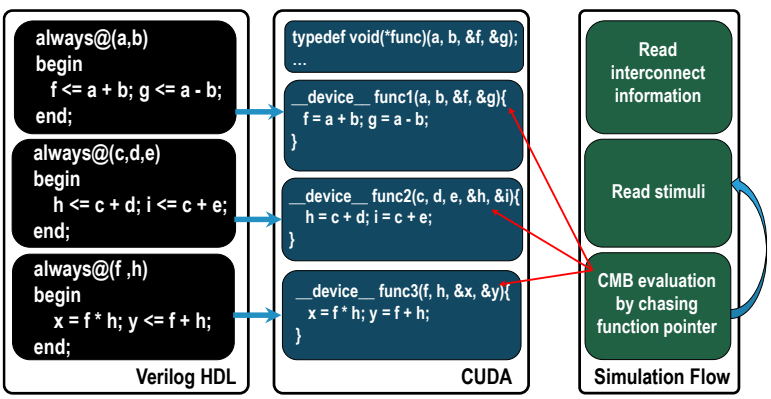


Fig. 5.25 GPU accelerated RTL simulation flow [Qian and Deng 2011]

5.4.3 SPICE Simulation

SPICE [Nagel 1995] simulation is the ultimate way to verify the correctness of circuit functioning. It is an indispensable step for any analog/RF and custom circuit designs. SPICE integrates a set of efficient procedures to simulate the non-linear time-varying behavior of a circuit. The simulation

process is organized into 4 tasks. First, the circuit is formulated as a set of equations in the matrix form for modified nodal analysis (MNA). A device evaluation operation is performed to fill the MNA matrix with respective device parameters. This step prepares data for the remaining 3 steps to compute. Second, the time-varying circuit behavior is derived by numerical integration techniques on the non-linear circuit model. Third, the non-linear circuit model is solved with a Newton-Raphson (NR) based iterative method. Fourth, each iteration of NR needs to solve a linear system of equations.

The solution of the linear equations is usually considered as the bottleneck of SPICE simulation [Ren et al. 2012]. MNA matrices are sparse and usually solved by a direct solver. During SPICE simulation of a circuit, the factorization pattern of the underlying MNA matrix is fixed and can be reused by multiple iterations. Therefore, the direct solver have a clear performance advantage over the iterative solvers. On the other hand, the device evaluation step can also be time-consuming in certain flows [Gulati et al. 2009].

Gulati et al. [2009] proposed a parallelization scheme for the above device evaluation problem in step 1. Although MNA matrices can be sparse, its distribution of non-zeros are fixed and known. Hence, copying the parameters of device models to MNA matrices can be a structured problem, which is explained in Section 4.7.

Ren et al. [2012] pioneered in developing GPU accelerated sparse LU factorization based solvers. The corresponding work is already detailed in sub-section 4.3.7. More research efforts are needed to construct robust and efficient direct solvers for sparse matrices.

6. CONCLUSION AND FUTURE WORK

In the previous chapters, we reviewed recent research works on accelerating EDA applications with GPGPU-enabled graphics processors. An emphasis is put on extracting and coordinating parallelism to design GPU-friendly parallel algorithms. We started by surveying the architectures and programming models of modern GPUs. Next we extended a taxonomy proposed by Catanzaro, Keutzer, and Su [2008] to classify fundamental computing patterns that are pervasive in EDA applications. The GPU based algorithms and implementations for these computing patterns were then examined in Chapter 4. Chapter 5 provided an overview on the usage of GPUs in expediting time-consuming EDA applications. Special attention was paid on how to apply the GPU-accelerated computing patterns to compose EDA tools and flows.

Based on the survey, it can be concluded that GPUs have a strong potential to shorten the run time of EDA applications. The superior computing power of GPUs can be released through carefully designed data-parallel algorithms and highly tuned implementations. In fact, the existing works of GPGPU based EDA applications already constitute a virtually complete VLSI design tool-chain.

On the other hand, we still see slow progresses in the EDA industry to deploy GPUs in mainstream design tools. The relative reluctance is due to a few reasons outlined as follows.

- GPGPU is a relatively new concept and GPU microarchitectures have not stabilized yet. Even on hardware from the same vendor, a piece of highly tuned code on one generation of GPU may need to be re-optimized for the next generation. Therefore, it may still be too early to invest R&D efforts on GPUs.
- With the emergence of OpenCL programming language, it is now feasible to achieve cross-platform compatibility. However, GPGPU programmers have to perform intensive tuning by taking into account hardware details so as to develop efficient GPU code. Such a fact suggests that

platform-specific optimization is still necessary even with a compatible GPU programming language.

- Current GPUs work as an accelerator for CPU. Data have to be first loaded by CPU from hard-drives to main memory and then sent to GPU for processing. As a result, data transfers become the bottleneck of many applications. A critical optimization is to minimize the data transfer behaviors, especially when a GPGPU program involves fine-grain cooperation between CPU and GPU. In the future, the problem can be mitigated because future GPU may integrate one or more CPU cores to host an operating system.
- With current programming technology, GPU programmers have to perform intensive hand-tuning to build highly efficiently programs. It is thus a formidable mission to port the huge code base of EDA software to GPU. Meanwhile, modern EDA tools deliver their power by working in a highly integrated framework. Porting a single EDA application to GPU may not bring significant performance advantage to the whole flow.
- Non-determinism is an inherent problem of multithreading [Lee 2006]. It is actually a common concern to both multi-core CPUs and GPUs. Due to the massive number of states involved in multithreaded execution on multiple cores, it is virtually impossible to maintain exactly the same operation environment for each run of a multithreaded program. Therefore, different runs may lead to (often slightly) diverse results. For IC design applications, such variations are only acceptable if they do not lead to results that cannot be reproduced. Otherwise, an irreproducible design implementation generated by a time-consuming EDA application makes debugging substantially more difficult, if not impossible. Recently, the computer architecture research community is working toward deterministic GPU microarchitectures [Jooybar et al. 2013].

Despite the above challenges, we believe GPUs will have to play an essential role in future EDA tools simply because of

their superior computing power. The EDA research community should consider the following routes to expedite the adventure toward true GPU-accelerated EDA computing.

Pattern based parallelization: The overwhelming complexity of EDA tools makes it infeasible to parallelize the whole EDA on GPU in on an application-by-application base. Instead, a feasible approach to focus on a few key computing patterns and develop highly efficient solutions for them. Such patterns can be shared by a large number of EDA applications.

Open-source research platform: Open-source EDA platforms like OpenEDA [Si2 2004] have proven to be critical to improve the quality of EDA research by offering well-designed reference algorithms, highly-tuned coding implementations and real-world experimental data. Current GPU-related EDA research, on the other hand, tends to perform experiments and comparison in a less standard manner. Such a fact makes harder to evaluate the effectiveness of proposed techniques. It is thus highly desirable to provide a unified GPU-oriented, open-source EDA research platform to standardize the research results.

Point tools for extremely time-consuming applications: Certain EDA tools can be operating in a relatively standalone fashion by using standard file formats for I/O operations. Logic simulators are typical tools in this category. Developing such point tools will not need an overhaul of the foundation of EDA software. Also considering the time complexity of logic simulation, it naturally follows that commercial GPU-accelerated EDA tools will first appear in this application niche (e.g., [Rocketick 2012]).

Automatic parallelization tools: With the prevalence of multi-core and many-core platforms, a new surge of research efforts for automatic parallelization is already emerging. A large body of research has been dedicated to ease the difficulty programming and fine-tuning GPU software. One approach to tradeoff programming effort with execution efficiency is the directive based parallelization. Relatively stable tools in this category have already been available (e.g., OpenHMPP [CAPS

2007] and OpenACC [OpenACC 2011]). Another approach is to automatically optimize various aspects of GPU code, namely, configuration of parallelism (e.g., [Ryoo 2008]), compiler based optimization (e.g., [Yang et al. 2010]), and data transfer (e.g., [Bauer et al. 2011; Jablin et al. 2011]). Various auto-tuning mechanism have also been proposed so that a GPU program can determine its optimal configuration according to the running environment (e.g., [Choi et al. 2010; Kurzak 2012]). The third, a more aggressive approach, is to synthesize parallel programs and its parallel execution schedule as well as the mapping to computing resources from a high level algorithmic description. Early frameworks for such a purpose have appeared (e.g., [Catanzaro et al. 2011; Han and Abdelrahman 2011]). Although it is still in their infancy, automatic parallelization is essential because the complexity of future parallel programs may finally become too complex for hand-crafted optimization. Of course, it is unlikely the automatic parallelization problem can be solved purely with compiler techniques. Instead, a synergy of computer architecture, compiler technology, and programming model is required to meet the ultimate goal. Meanwhile, it also should be noted that many EDA algorithms can be borrowed by automatic parallelization and optimization tools. The developing of parallel EDA software and automatic parallelization techniques should be performed as an interactive process.

7. ACKNOWLEDGEMENTS

This work was partially supported by China National Science Foundation under contract number 61272085 and NVIDIA Professor Partnership Awards. The authors also acknowledge hardware donation from NVIDIA. We thank the anonymous reviewers for their constructive feedback and thorough evaluation.

REFERENCES

Abramovici, M., Menon, P. R., and Miller, D. T. 1983. Critical path tracing - an alternative to fault simulation. In *Proc. of Design Automation Conference*. 214–220.

Agullo, E., et al. 2011. LU Factorization for accelerated-based systems. *In Proc. of International Conference on Computer Systems and Applications*. 217-224.

AMD. 2009. ATI stream computing - technical overview. (http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf).

AMD. 2011. Graphics Core Next architecture. http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx.

AMD. 2012. AMD Radeon™ and AMD FirePro™ graphics cards. http://www.amd.com/us/products/Pages/graphics.aspx

Anderson, E., et al 1990. LAPACK: a portable linear algebra library for high-performance computers. *In Proc. of Conference on Supercomputing*. 2-11.

Anderson M., Ballard G., Demmel J., and Keutzer K. 2010. *Communication-avoiding QR decomposition for GPUs*. *Technical Report No. UCB/EECS-2010-131*.

Ashlock, D. 2006. Evolutionary computation for modeling and optimization. Springer. ISBN 0-387-22196-4.

Augonnrt, C., Thibault, S. and Namyst, R. 2010. StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. *INRIA, Technique Report 7240*. http://hal.archives-ouvertes.fr/inria-00467677.

Baboulin, M., Donfack, S., Dongarra, J., Grigori, L., Remy, A., and Tomov, S. 2012. A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines. *In Proc. of International Conference on Computational Science.* 17-26.

Baggio, D. L. 2007. GPGPU based image segmentation livewire algorithm implementation. *Master Thesis of Technological Institute of Aeronautics, S˜aoJos´e dos Campos*.

Bailey, M. L., Brinerjr., J. V., Chamnerlain, R. D. 1994. Parallel logic simulation of VLSI systems. *ACM Computing Survey*. 26(3):255–294.

Banzhaf, W., Nordin, P., Keller, R., Francone, F., 1998. Genetic programming – an introduction. San Francisco, CA: Morgan Kaufmann. ISBN 978-1558605107.

Bauer, M., Cook, H., and Khailany, B. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. *In Proc. of Conference on Supercomputing*.

Barrachina S., Castillo, M., Igual, F. D., Mayo, R., Quintana-Ortí, E. S., and Quintana-Ortí, Gregorio. 2009. Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurrency and Computation: Practice and Experience*. 21(18):2457-2477.

Barret, R. et al. 1994. Templates for the solution of linear systems. 2nd Edition. SIAM.

Baskaran, M. M., and Bordawekar, R. 2009. Optimizing sparse matrix–vector multiplication on GPUs. *IBM Technical report RC24704*.

Beamer, S., Asanovi´c, K., and Patterson, D. 2012. Direction-optimizing breadth-first search. *In Proc. of International Conference on High Performance Computing, Networking, Storage and Analysis*.

Bell, N. and Garland, M. 2008. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report.

Bell, N. and Garland, M. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *In Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*.

Ben-Asher, Y., Egozi, D., and Schuster, A. 1989. 2-D SIMD algorithms in the perfect shuffle networks. *In Proc. of International Symposium of Computer Architecture*. 88-95.

Berkeley Logic Synthesis and Verification Group. 2005. ABC: A system for sequential synthesis and verification. http://www.cad.eecs.berkeley.edu/~alanmi/abc.

Biere, A., Heule, M., Van Maaren, H., and Walsh, T. 2009. Handbook of satisfiability. IOS Press.

BLAS. 2008. Basic linear algebra subprograms. http://www.netlib.org/blas/

Bleiweiss, A. 2008. GPU accelerated pathfinding. *In Proc. of Symposium on Graphics Hardware*. 65-74.

Bleiweiss, A. 2009. Multi agent navigation on the GPU. *In Proc. of Game Developers Conference.*

Blelloch, G. E. 1990. Vector models for data-parallel computing. MIT Press.

Blythe, D. 2008. Rise of the Graphics Processor. *Proceedings of IEEE*. 96(5):761– 778.

Bombieri, N., Fummi, F., and Guarnieri, V. 2012a. FAST-GP: an RTL functional verification framework based on fault simulation on GP-GPUs. *In Proc. of Design, Automation, and Test in Europe (DATE) Conference*. 562–565.

Bombieri, N., Vinco, S., Bertacco, V., and Chatterjee, D. 2012b. SystemC simulation on GP-GPUs: CUDA vs. OpenCL. *In Proc. of International Conference on Hardware/Software Codesign and System Synthesis*. 343-352.

Bordoloi, U. D., and Chakraborty, S. 2010. GPU-based acceleration of system-level design tasks. *International Journal of Parallel Programming*. 38(3-4):225-253.

Boukedjar, A., Lalami, M.E., and El-Baz, D. 2012. Parallel branch and bound on a CPU-GPU system. *In Proc. of Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. 392 -398

Briggs, W. 1987. A multigrid tutorial. SIAM Press.

Brown, S. 2010. Cadence contributes ESL methodology to TSMC reference flow.

http://www.cadence.com/community/blogs/sd/archive/2010/06/11/system-realization-costs-seen-as-critical-barrier-to-ic-development-and-potentially-impacting-foundry-business.aspx.

Bryant, R. E. 1977. Simulation of packet communications architecture computer system. *MIT Technical Report MITLCS-TR-188*.

Bryant, R.E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*. C-35(8):677 – 691.

Buatois, L., Caumon, G., and Levy, B. 2009. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*. 24(3):205-223.

Buck, I., et al. 2004. Brook for GPUs: stream computing on graphics hardware. *In Proc. of SIGGRAPH*. 777-786.

Buluç, A., Gilbert, J. R., and Budak, C. 2010. Solving path problems on the GPU. *Parallel Computing*. 36(5-6):241-253.

Burton, M., Morawiec, A. (Editor). 2006. Platform based design at the electronic system level: industry perspectives and experiences. Springer. 2006 edition. ISBN 978-1-4020-5138-8.

Butenhof, D. R. 1997. Programming with POSIX threads. Addison-Wesley Professional. ISBN-10: 0201633922.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*. 35(1): 38–53.

Cadence. 2008. Virtuoso accelerated parallel simulator . http://www.cadence.com/products/cic/accelerated_parallel/pages/default.aspx.

Caldwell, A. E., Kahng, A. B., and Markov, I. L. 2000. Can recursive bisection alone produce routable placements? *In Proc. of Design Automation Conference*. 693-698.

CAPS. 2007. OpenHMPP directives. http://www.caps-entreprise.com/openhmpp-directives/

Catanzaro, B., Garland, M., and Keutzer, K. 2011. Copperhead: compiling an embedded data-parallel language. *In Proc. of Symposium on Principles and Practice of Parallel Programming*. 47-56.

Catanzaro, B., Keutzer, K., and Su, B.-Y. 2008. Parallelizing CAD: a timely research agenda for EDA. *In Proc. of Design Automation Conference*. 12-17.

Chakroun, I., Mezmaz, M., Melab, N., and Bendjoudi, A. 2012. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*. 25(8): 1121-1136.

Chan, T., Cong, J., and Sze, K. 2005. Multilevel generalized force-directed method for circuit placement. *In Proc. of International Symposium on Physical Design*. 185–192.

Chandy, K. M., and Misra, J. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Software Engineering*. SE-5(5):440–452.

Chapman, B., Jost, G., and van der Pas, R. 2007. Using OpenMP: portable shared memory parallel programming. The MIT Press. ISBN: 9780262533027.

Chatterjee, D., and Bertacco, V. 2010. EQUIPE: parallel equivalence checking with GPUs. *In Proc. of International Workshop on Logic & Synthesis*. 486-493.

Chatterjee, D., DeOrio, A., and Bertacco, V. 2007. High-performance gate-level simulation with GP-GPUs. *In Proc. of Design, Automation, and Test in Europe (DATE) Conference*. 1-3.

Chatterjee, D., DeOrio, A., and Bertacco, V. 2009. Event-driven gate-level simulation with GP-GPUs. *In Proc. of Design Automation Conference*. 557-562

Chatterjee, D., DeOrio, A., and Bertacco, V. 2011. Gate-level simulation with GPU computing. *ACM Transactions on Design Automation of Electronic Systems. 16(3)*.

Chen, D., and Singh, D. 2010. Parallelizing FPGA technology mapping using graphics processing units (GPUs). *In Proc. of International Conference on Field Programmable Logic and Applications*. 125-132.

Choi, J. W., Singh, A. Vuduc, and R. W. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *In Proc. of Principles and Practice of Parallel Computing*. 115-126.

Choong, A., Beidas, R. and Zhu, J. 2010. Parallelizing simulated annealing-based placement using GPGPU. *In Proc. of International Conference on Field Programmable Logic and Applications*.

Chu, K.-W., Deng, Y., and Reinitz, J. 1999. Parallel simulated annealing by mixing of states. *Journal of Computational Physics*. 148. 646–662.

Clark, J. H. 1982. The geometry engine: A VLSI geometry system for graphics. *In Proc. of Annual Conference on Computer Graphics and Interactive Techniques*. 127-133.

Clarke, E. M., and Grumberg, O. 1987. Avoiding the state explosion problem in temporal logic model checking. *In Proc. of Symposium on Principles of Distributed Computing*. 294-303.

Cong, J., and Zou, Y. 2009. Parallel multi-level analytical global placement on graphics processing units. *In Proc. of International Conference on Computer Aided Design*. 681-688.

Cook, R. L., Carpenter L., and Catmul, E. 1987. The Reyes image rendering architecture. *In Proc. of Conference on Computer Graphics and Interactive Techniques*. 95-102.

Cormen, T. H., Leiserson, C. E., Riverst, R. L., and Stein, C. 2001. Introduction to algorithms, 2nd Edition. MIT Press.

CUDPP. 2010. CUDA data-parallel primitives library. http://code.google.com/p/cudpp/

Darema, F. 2001. SPMD model: past, present and future. Recent Advances in Parallel Virtual Machine and Message Passing Interface. *In Proc. of European PVM/MPI Users' Group Meeting*. Lecture Notes in Computer Science 2131.1.

Deng, Y., and Mu, S. 2008. The potential of GPUs for VLSI physical design automation. In *Proc. of International Conference on Solid-State and Integrated-Circuit Technology*. 2272-2275.

Deng, Y., Wang, B., and Mu, S. 2009. Taming irregular EDA applications on GPUs. *In Proc. of International Conference on Computer Aided Design*. 539-546

Dean, J., and Ghemawat, S. 2004. MapReduce: Simplified data processing on large clusters. *In Proc. of Symposium on Operating System Design and Implementation*. 137-149.

Dongarra, J., Eijkhout, V., and Luszczek, P. 1998. Recursive approach in sparse matrix LU factorization. *Scientific Programming*. 9(1):51-60.

Dotsenko, Y., Govindaraju, N. K., Sloan, P.-K., Boyd, C. and Manferdelli, J. 2008. Fast scan algorithms on graphics processors. *In Proc. of International Conference on Supercomputing*. 205–213.

Eisenmann, H. and Johannes, F. M. 1998. Generic global placement and floorplanning. *In Proc. of Design Automation Conference*. 269-274.

Fatahalian K., and Houston, M. 2008. GPUs: a closer look. *ACM Queue*. 6(2):18-28.

Feng, Z. and Li, P. 2008. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. *In Proc. of International Conference on Computer Aided Design*. 647-654.

Feng, Z., and Li, P. 2010. Fast thermal analysis on GPU for 3D-ICs with integrated microchannel cooling. *In Proc. of International Conference on Computer Aided Design*. 551-555.

Feng, Z., and Zeng, Z. 2010. Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis. *In Proc. of Design Automation Conference*. 661-666.

Fisher, J. A. 1983. Very long instruction word architecture and the ELI-512. *In Proc. of International Symposium on Computer Architecture*. 140-150.

Folley, T. 2008. Parallel programming on Larrabee. *Special Tutorial on SIGGRAPH*. 2008.

Fujimoto, R. M. 2000. Parallel and distributed simulation systems. Wiley-Interscience. ISBN: 0471183830

Fung, W. L., Sham, I., Yuan, G., and Aamodt T. M., 2007. Dynamic warp formation and scheduling for efficient GPU control flow. *In Proc. of International Symposium on Microarchitecture*. 407-420.

Galiano, V., Migalló, N. H., MigallóN, V., and PenadéS, J. 2012. GPU-based parallel algorithms for sparse nonlinear systems. *Journal of Parallel and Distributed Computing*.72(9):1098-1105.

Galoppo, N. el al. 2005. LU-GPU: efficient algorithms for solving dense linear systems on graphic hardware. *In Proc. of ACM/IEEE Conference on Supercomputing*. 3-14.

Garland, M. 2008. Sparse matrix computations on many-core GPUs. *In Proc. of Design Automation Conference*. 2-6.

Ghuloum, A. et al. 2007. Ct: a flexible parallel programming model for tera-scale architectures. Intel White Paper.

http://download.intel.com/pressroom/kits/research/Flexible_Parallel_Programming_Ct.pdf.

Gilbert, J. R., and Peierls, T. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal of Scientific Statistical Computing*. 9(5):862–874.

Graph500. 2010. The graph 500 list. http://www.graph500.org/.

Gratton, S. 2009. Cholesky factorization in CUDA. http://www.ast.cam.ac.uk/~stg20/cuda/cholesky/index.html.

Greengard, L., and Rokhlin, V. 1987. A fast algorithm for particle simulations. Journal of Computing Physics. 73(2):325–348.

Grewe, D., and Lokhmotov, A. 2011. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. *In Proc. of Workshop on General Purpose Processing on Graphics Processing*.

Grimes, R., Kincaid, D., and Young, D. 1979. ITPACK 2.0 user's guide. Technical Report CNA-150. Center for Numerical Analysis, University of Texas.

Gu, Z., Wang, J., Dick, R. P., and Zhou, H. 2005. Incremental exploration of the combined physical and behavioral design space. *In Proc. of Design Automation Conference*. 208–213.

Gulati, K. and Khatri, S. P. 2008. Towards acceleration of fault simulation using graphics processing units. *In Proc. of Design Automation Conference*. 822-827.

Gulati, K., Croix, J. F., Khatri, S. P., and Shastry, R. 2009. Fast circuit simulation on graphics processing units. *In Proc. of Conference on Asia and South Pacific Design Automation*. 403-408.

Gulati, K., Khatri, S.P. 2009b. Accelerating statistical static timing analysis using graphics processing units. *In Proc. of Conference on Asia and South Pacific Design Automation*. 260-265.

Gumerov, N., and Duraiswami, R.. 2008. Fast multipole methods on graphics processors. *Journal of Computing Physics*. 227(18):8290–8313.

Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. 2001. FLAME: formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)*. 27(4):422-455.

Guo, P., et al. 2011. A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on GPUs. *In Proc. of TeraGrid*.

Hamada, T. et al. 2009. 42 TFLOPS hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. *In Proc. of Conference on High Performance Computing Networking, Storage and Analysis*.

Han, T. D., and Abdelrahman, T. S. 2011. hiCUDA: high-level GPGPU programming. *IEEE Transactions onParallel and Distributed Systems.* 22(1):78-90.

Han, Y., Ancajas, D. M., Chakraborty, K., and Roy, S. 2011a. Exploring high throughput computing paradigm for global routing. I*n Proc. of International Conference on Computer Aided Design*. 298-305.

Han, Y., Chakraborty, K., Roy, S., and Kuntamukkala, V. 2011b. Design and implementation of a throughput-optimized GPU floorplanning algorithm. *ACM Transactions on Design Automation of Electronic Systems*. 16(3).

Harish, P., and Narayanan, P. J. 2007. Accelerating large graph algorithms on the GPU using CUDA. *In Proc. of High Performance Computing (HiPC)*. 197-208.

Harish, P., Vineet, V., and Narayanan, P. J. 2009. Large graph algorithms for massively multithreaded architectures. *IIIT Technical Report IIIT/TR/2009/74*. http://web.iiit.ac.in/~vibhavvinet/Publications/GraphAlgos_TechRep.pdf

Harris, M. 2007. Parallel prefix sum (scan) with CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf.

Harris, M. 2008. Optimizing parallel reduction in CUDA. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/reduction/doc/reduction.pdf

He. B. et al. 2008. Mars: a MapReduce framework on graphics processors. *In Proc. of Parallel Architectures and Compilation Techniques*. 260-269.

He, Z., and Hong, B. 2010. Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU hybrid platforms. *In Proc. of International Parallel and Distributed Processing Symposium*. 1-10.

Heath, M. T., Ng, E., and Peyton, B. W. 1991. Parallel algorithms for sparse linear systems. *SIAM Review*. 33. 420-460,

Helfenstein, R., and Koko, J. 2012. Parallel preconditioned conjugate gradient algorithm on GPU. *Journal of Computational and Applied Mathematics*. 236(15):3584-3590.

Herlihy, M. and Shavit, N. 2008. The art of multiprocessor programming, Morgan Kaufmann Publishers. Burlington.

Hong, S. et al. 2011. Accelerating CUDA graph algorithms at maximum warp. *In Proc. of Symposium on Principles and Practice of Parallel Programming*. 267–276.

Hong, S., Oguntebi, T., and Olukotun, K. 2011. Efficient parallel graph exploration on multi-Core CPU and GPU. *In Proc. of Parallel Architectures and Compilation Techniques*. 78-88.

Hsu, C.-J., Pino, J. L., and Bhattacharyya, S. S. 2008. Multithreaded simulation for synchronous dataflow graphs. *In Proc. of Design Automation Conference*. 331-336.

Huang, J. 2010. Keynote Speech. *Mini GPU Technology Conference*. Beijing.

Huijs, C. 1996. A graph rewriting approach for transformational design of digital systems. *In Proc. of EUROMICRO Conference*. 177-184.

Humphrey, J. R., et al. 2010. CULA: hybrid GPU accelerated linear algebra routines. *In Proc. of SPIE Modeling and Simulation for Defense Systems and Applications*. *V*770502-770502-7.

Hussein, M., Varshney, A., and Davis, L. 2007. On implementing graph cuts on cuda. *In Proc. of the First Workshop on General Purpose Processing on Graphics Processing Units*.

IEEE. 2004. The open group base specifications Issue 6. IEEE Standard 1003.1, 2004 Edition. (http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html)

IEEE. 2011. SystemC language – 2011. http://standards.ieee.org/getieee/1666/download/1666-2011.pdf

Intel. 2010. Intel® array building blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks

Intel. 2012. The Intel ® Xeon Phi ™ coprocessor 5110P highly-parallel processing for unparalleled discovery. http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/high-performance-xeon-phi-coprocessor-brief-2.pdf.

Jablin, T. B., et al. 2011. Automatic CPU-GPU communication management and optimization. *In Proc. of Conference on Programming Language Design and Implementation*. 142-151.

Jantsch, A. 2004. Modeling embedded systems and SoCs. Morgan Kaufmann.

JEDEC. 2009. JEDEC standard: GDDR5 SGRAM. http://www.jedec.org/standards-documents/docs/jesd212.

JEDEC. 2012. JEDEC standard: DDR4 SDRAM. http://www.jedec.org/standards-documents/docs/jesd79-4.

Jooybar, H., Fung, W. W. L., O'Connor, M. Devietti, J., and Aamodt, T. M. 2013. GPUDet: a deterministic GPU architecture. *In Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*. 1-12.

Jost, T., Contassot-Vivier, S., and Vialle, S. 2009. An efficient multi-algorithms sparse linear solver for GPUs. *In Proc. of International Conference on Parallel Computing (ParCo)*.

Jung, J., and O'Leary, D. P. 2006. Cholesky decomposition and linear Programming on a GPU. *In Proc. of Workshop on Edge Computing Using New Commodity Architectures*.

Kapoor, R., Adan, M., and Schaffer, L. 2004. Achieving optimal performance scalability for physical verification. http://www.synopsys.com/Tools/Implementation/PhysicalVerification/CapsuleModule/hercules_achiv_wp.pdf

Karypis, K., Aggarwal, R., Kumar, V., and Shekhar, S. 1997. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on VLSI Systems*. 7(1):69-79.

Katz, G. J., and Kider, J. T. Jr. 2008. All-pairs shortest-paths for large graphs on the GPU. *In Proc. of Symposium on Graphics Hardware*. 47–55.

Keller, J., Keßler, C., Träff, J. 2001. Practical PRAM programming. John Wiley and Sons. ISBN 0-471-35351-5.

Kerr A., Campbell D., and Richards M. 2009. QR Decomposition on GPUs, *In Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*. 71-78.

Keutzer, K. 1987. DAGON: technology binding and local optimization by DAG matching. *In Proc. of Design Automation Conference*. 341-347.

Khajeh-Saeed, A., Poole, S., and Perot, J. B. 2010. Acceleration of the Smith–Waterman algorithm using single and multiple graphics processors. *Journal of Computational Physics*. 229(11):4247–4258.

Khabou A., Demmel J. W., Grigori L., and Gu, M. 2012. LU factorization with panel rank revealing pivoting and its communication avoiding version. *UCB Technical Report No. UCB/EECS-2012-15*.

Khronos Group. 2012. OpenCL - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

Kim K., Eijkhout V., and Geijn R. A. 2012. Dense matrix computation on a heterogeneous architecture: a block synchronous approach. *TACC Technique Report TR-12-04*.

Kleinhans, G., Sigl, F. J., and Antreich, K. 1991. Gordian: VLSI placement by quadratic programming and slicing optimization. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*. 10(3):356-365.

Kochte, M. A., Schaal, M., Wunderlich, H.-J., and Zoellin, C. G. 2010. Efficient fault simulation on many-core processors. *In Proc. of Design Automation Conference*. 380-385.

Kurzak J., Tomov S., and Dongarra J. 2010. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice & Experience*. 22(1):15-44.

Kurzak, J. Tomov S., and Dongarra J.2012. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*. 23(11):2045-2057.

Langdon, W. B. 2008. A fast high quality pseudo random number generator for graphics processing units. *In Proc. of Congress on Evolutionary Computation*. 459-465.

Lattner, C., and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *In Proc. of International Symposium on Code Generation and Optimization*. 75-86.

Lee, E. 2006. The problem with threads. *IEEE Computer*. 39(5):33-42.

Levinthal, A., and Porter, P. 1984. Chap - a SIMD graphics processor. *In Proc. of Conference on Computer graphics and interactive techniques*. 77-82.

Lewis, D. 1991. A hierarchical compiled code event-driven logic simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 10(6):726-737.

Li, C., Ding, C., and Shen, K. 2007. Quantifying the cost of context switch. *In Proc. of Workshop on Experimental Computer Science*.

Li, M., Hsiao, M. S. 2011. 3-D parallel fault simulation with GPGPU. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 30(10):1545-1555.

Li J., Li X., Tan G., Chen M., and Sun N. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. *In Proc. of International Conference on Supercomputing*. 377-386.

Lin, Y.-L. S. 2010. Essential issues in SOC design: designing complex Systems-on-Chip. Springer. Softcover reprint of hardcover 1st ed. 2006 edition.

Liu, J. 1986. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*. 3(4):327-342.

Liu, J. 1992. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review*. 34(1):82-109.

Liu, W. et al. 1999. BSIM3v3.2.2 MOSFET Model: Users' Manual.

Liu, Y., and Hu, J. 2009. A new algorithm for simultaneous gate sizing and threshold voltage assignment. *In Proc. of the International Symposium on Physical Design (ISPD)*.

Liu, Y., and Hu, J. 2011. GPU-based parallelization for fast circuit optimization. *ACM Transactions on Design Automation of Electronic Systems*. 16(3).

Liu, Y., Huang, W., Johnson, J., and Vaidya, S. 2006. GPU accelerated Smith-Waterman. *In Proc. of International Conference on Computational Science*. Lecture Notes in Computer Science Volume 3994. 188-195.

Liu, X., Liu, Z., Tan, S. X.-D., and Gordon, J. A. 2012. Full-chip thermal analysis of 3D ICs with liquid cooling by GPU-accelerated GMRES method. *In Proc. of IEEE/ACM International Symposium on Low-Power Electronics and Design*. 123-128.

Liu, Y., Schmidt, B., and Maskell, D. L. 2010. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*. 3(93).

Low, Y. et al. 2010. GraphLab: a new parallel framework for machine learning. *In Proc. of Conference on Uncertainty in Artificial Intelligence*.

Lu, Y., Zhou, H., Shang, L., and Zeng, X. 2009. Multicore parallel min-cost flow algorithm for CAD applications. *In Proc. of Design Automation Conference*. 832-837.

Luo, L., Wong, M., and Hwu, W.-M. 2010. An effective GPU implementation of breadth-first search. *In Proc. of Design Automation Conference*. 52-55.

Van Luong, T., Melab, N., and Talbi, E. G. 2010. Large neighborhood local search optimization on graphics processing units. *Proc. of International Parallel and Distributed Processing Symposium*. 1-8.

Martin, G., Bailey, and B., Piziali, A. 2007. ESL design and verification: a prescription for electronic system level methodology. Morgan Kaufmann; 1st edition.

Matam, K. K., and Kothapalli, K. 2011. Accelerating sparse matrix vector multiplication in iterative methods using GPU. *In Proc. of International Conference on Parallel Processing*.

Matsumoto, K., Nakasato, N., and Sedukhin, S.G. 2011. Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. *In Proc. of High Performance Computing and Communications (HPCC)*. 145 – 152.

Mattson, T. G., Sanders, B. A., and Massingill, B. L. 2004. Patterns for parallel programming. Addison-Wesley Professional.

McCool, M. D. 2008. Scalable programming models for massively multicore processors. *Proceedings of IEEE*. 96(5):816-831.

McGlaun, S. 2009. GPU market shows impressive growth in Q2 2009. SlipperyBrick. http://www.slipperybrick.com/2009/07/gpu-market-shows-impressive-growth-in-q2-2009/.

Melab, N., Chakroun, I., Mezmaz, M., and Tuyttens, D. 2012. A GPU-accelerated branch-and-bound algorithm for the flow-shop scheduling problem. *In Proc. of International Conference on Cluster Computing*.

Meng, J., Tarjan, D., and Skadron, K. 2011. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *In Proc. of International Symposium on Computer Architecture*. 235-246.

Merrill, D., Garland, M., and Grimshaw, A. 2012. Scalable GPU graph traversal. *In Proc. of the Symposium on Principles and Practice of Parallel Programming*. 117-128.

Meyer, U., Sanders, P. 1998. Delta-stepping: a parallel single source shortest path algorithm. *In Proc. of Annual European Symposium on Algorithms*. 393-404.

Microsoft. 2000. DirectX 11. http://windows.microsoft.com/zh-CN/windows7/products/features/directx-11.

Muller, D. 2006. Optimizing yield in global routing. *In Proc. of International Conference on Computer Aided Design*. 480-486.

Nam, G.-J., Alpert, J. C., and Villarrubia, P. G. 2007. ISPD 2005/2006 placement benchmarks. Modern Circuit Placement. Springer US.

Nabors, K., and White, J. 1991. FastCap: a multipole accelerated 3-D capacitance extraction program. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*. 10. 11. 1447 –1459.

Nagel, L. 1995. SPICE: A computer program to simulate computer circuits. *University of California, Berkeley UCB/ERL Memo M520*.

Nanjundappa, M., Patel, H. D., Jose, B. A., and Shukla, S. K. 2010. SCGPSim: a fast SystemC simulator on GPUs. *In Proc. of Asia and South Pacific Design Automation Conference*. 149-154.

Narasiman, V., Shebanow, M., Lee, C. J., Miftakhutdinov, R., Mutlu, O., and Patt, Y. N. 2011. Improving GPU performance via large warps and two-level warp scheduling. *In Proc. of International Symposium on Microarchitecture*. 308-317.

Nath R., Tomov S., and Dongarra J. 2010. An improved magma GEMM for Fermi GPUs. *Technical Report 227*. LAPACK Working Note.

Naumov, M., Chien, L. S., Vandermersch, and P., Kapasi, U. 2010. CUSPARSE Library. *GPU Technology Conference*.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. 2008. Scalable parallel programming with CUDA. *ACM Queue*. 6(2):40–53.

Nie, J. 2009. Evaluating the potential of Intel Ct technology for EDA Computing. *Bachelor thesis*. Tsinghua University.

NVIDIA. 2006. NVIDIA GeForce 8800 GPU architecture overview. Technical Brief.

NVIDIA. 2007a. NVIDIA CUDA Compute Unified Device Architecture programming guide. Version 1.0.

NVIDIA. 2007b. PTX: Parallel Thread Execution ISA. Version 1.0.

NVIDIA. 2007c. CUBLAS. https://developer.nvidia.com/cublas.

NVIDIA. 2007d. CUFFT. https://developer.nvidia.com/cufft.

NVIDIA. 2009. NVIDIA's next generation CUDATM compute architecture: Fermi™. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

NVIDIA. 2012a. NVIDIA's next generation CUDATM compute architecture: Kepler™ GK110. (http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf).

NVIDIA. 2012b. CUDA GPUs. https://developer.nvidia.com/cuda-gpus

Okuyama, T., Ino. F., and Hagihara, K. 2008. A task parallel algorithm for computing the costs of all-pairs shortest paths on the CUDA-compatible GPU. *In Proc. of International Symposium on Parallel and Distributed Processing with Applications*. 284-291.

Onbaşoğlu, E., and Özdamar, L. 2001. Parallel simulated annealing algorithms in global optimization. *Journal of Global Optimization*. 19(1):27-50.

OpenGL. 2012. The OpenGL graphics system: a specification. http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf.

OpenACC. 2011. The OpenACC™ application programming interface. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

OSCI. 2007. SystemC 2.2. http://www.accellera.org/members/download_files/check_file?agreement=systemc-2_2-draft.

Osipov, V., Sanders, P. and Singler, J. 2009. The Filter-Kruskal Minimum Spanning Tree Algorithm. *In Proc. of Workshop on Algorithm Engineering and Experiments*.

Otellini, P. 2011. Keynote speech. Intel Developer Forum. 2011

Owens, J. D., Luebke, D., Govindaraju, N., Houston, M., Krüger, J., Lefohn, A. E., and Purcell, T. A. 2005. A survey of general-purpose computation on graphics hardware. *Eugographics: State of the Art Reports*. 21-51.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J.E., and Phillips, J. C. 2008. GPU computing. *Proceedings of IEEE*. 96(5):879-899.

Pichel, J.C., Singh, D.E., and Carretero, J. 2008. Reordering algorithms for increasing locality on multicore processors. *In Proc. of the IEEE International Conference on High Performance Computing and Communications*. 123–130.

Pichel, J. C., Rivera, F. F., Fernandez, M., and Rodríguez, A. 2012. Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems*. 36(2):65–77

Pinel, F., Dorronsoro, and B., Bouvrya, P. 2013. Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel Distributed Computing*. 73(1):101–110.

Pingali, K. et al. 2011. The tao of parallelism in algorithms. *In Proc. of Programming Language Design and Implementation*. 12-25.

Pinto, C., et al. 2011. GPGPU-accelerated parallel and fast simulation of thousand-core platforms. *In Proc. of International Symposium on Cluster, Cloud and Grid Computing*.

PCI-SIG. 2002. PCI express base specification. Revision 1.0.

QEMU. Full system processor emulator. http ://wiki.qemu.org.

Qian, H., Deng, Y. 2011. Accelerating RTL simulation with GPUs. *In Proc. of International Conference on Computer Aided Design*. 687-693.

Qian, H., Deng, Y., Wang, B., Mu, S. 2012. Towards accelerating irregular EDA applications with GPUs. *Integration: the VLSI Journal*. 45(1): 46-60.

Raghav, S., Ruggiero, M., Atienza, D. 2010. Scalable instruction set simulator for thousand-core architectures running on GPGPUs. *In Proc. of International Conference on High Performance Computing and Simulation*.

Raghav, S., et al. 2012. Full System Simulation of Many-Core Heterogeneous SoCs using GPU and QEMU Semihosting. *In Proc. of GPGPU Workshop*.

Ramalingam, A., et al. 2006. An accurate sparse matrix based framework for statistical static timing analysis. *In Proc. of International Conference on Computer Aided Design*.

Ranger, C. et al. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. *In Proc. of International Symposium on High-Performance Computer Architecture*. 13-24.

Ren, L. et al. 2012. Sparse LU Factorization for Parallel Circuit Simulation on GPU. *In Proc. of Design Automation Conference*.

Rashinkar, P., Paterson, P., Singh, L. 2000. System-on-a-Chip verification: Methodology and Techniques. Springer. Edition 1.

Rhu, M., and Erez, M. 2012. CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. *In Proc. of International Symposium on Computer Architecture*. 61-71.

Rocketick. 2012. Harnessing the power of GPU. http://www.rocketick.com/technology/harnessing-the-power-of-gpu.

Rostrup, S., Srivastava, S., and Singhal, K. 2013. Fast and Memory-Efficient Minimum Spanning Tree on the GPU. *International Journal of Computational Science and Engineering*. 8(1):21-33.

Russell, S. J., Norvig, P. 2003. Artificial intelligence: a modern approach. Prentice Hall. 97-104.

Rutenbar, R. 2007. Next-generation design and EDA challenges: small physics, big systems, and tall tool-Chains. *In Proc. of Asia and South Pacific Design Automation Conference*.

Ryoo, S, et al. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *In Proc. of Principles and Practice of Parallel Programming*. 73-82.

Saad, Y. 2000. Iterative methods for sparse linear systems. SIAM.

Sapatnekar, S. 2004. Timing. Springer, 1st Edition. Ch. 5.

Sapatnekar, S. et al. 2008. Reinventing EDA with many-core processors. *In Proc. of Design Automation Conference*.

Savran, I., Bakos, J. D. 2010. GPU acceleration of near-minimal logic minimization. *In Proc. of Symposium on Application Accelerators in High-Performance Computing*.

Seiler, L. et al. 2008. Larrabee: a many-Core X86 architecture for visual computing. *ACM Transaction on Graphics*. 3(8):18:1-18.16.

Sen, A., Aksanli, B., and Bozkurt, M. 2010. Speeding up cycle based logic simulation using graphics processing units. *International Journal of Parallel Programming*. 39(5):639-661.

Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. 2007. Scan primitives for GPU computing. *In Proc. of Graphics Hardware*. 97-106.

Seoane, N., and Garcia-Loreiro, A. J. 2005. Study of parallel numerical Methods for semiconductor device simulation. *International Journal of Numerical Modeling: Electronic Networks, Devices, and Fields*. 19(1):15-32.

Sherwani, N. A. 1998. Algorithm for VLSI physical design automation. 3rd Edition. Springer.

Shi, J. et al. 2009. GPU friendly fast Poisson solver for structured power grid network analysis. *In Proc. of Design Automation Conference*. 178-183.

Shimpi, A. L., and Wilson, D. 2008. Intel's Larrabee architecture disclosure: a calculated first move. (http://www.anandtech.com/showdoc.aspx?i=3367)

Shapira, Y. 2009. Matrix-based multigrid: theory and applications. Springer.

Si2. 2004. OpenEDA. http://www.si2.org/openeda.si2.org/oso_news.php.

Siek, J. G., Lee, L.-Q., and Lumsdaine, A. 2002. Boost Graph Library, The: User Guide and Reference Manual. Addison-Wesley Professional.

Smalley, T. 2008. RV770: ATI Radeon HD 4850 & 4870 analysis. bit-tech.net. (http://www.bit-tech.net/hardware/graphics/2008/09/02/ati-radeon-4850-4870-architecture-review/1).

SMIC. 2013. SMIC-Cadence reference flow 2.1. http://www.smics.com/eng/design/reference_flows05.php

Song F., Tomov S., and Dongarra J. 2012. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *Proc. of ACM International Conference on Supercomputing*. 365-376.

Stephens, R. 1995. A survey of stream processing. *Acta Informatica*. 34(7):491-554.

Stroustrup, B. 1997. The C++ programming language. 3rd Edition. ISBN 0-201-88954-4.

Strzodka R. Accelerated ANSYS fluent: algebraic multigrid on a GPU. Available on http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/RobertStrzodka_Accelerated_ANSYS_Fluent_SC12.pdf.

Subramany, L. 2011. GPU based lithography simulation and OPC. *Master Thesis*. Department of Electrical and Computer Engineering. University of Massachusetts Amherst.

Suri, B., Bordolo, U. D., and Eles, P. 2012. A scalable GPU-based approach to accelerate the multiple-choice knapsack problem. *In Proc. of Design, Automation, and Test in Europe (DATE) Conference*. 1126-1129.

Synopsys. 2008a. The gold standard for accurate circuit simulation. http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSPICE/Pages/default.aspx.

Synopsys. 2008b. Synopsys unveils new IC compiler router delivering 10X speed-up. http://news.synopsys.com/index.php?s=43&item=575.

Tan, G., Li, L., Trieche, S., Phillips, E., Bao, Y., and Sun, N. 2011. Fast implementation of DGEMM on Fermi GPU. *In Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*.

Topaloglu, R. O. 2011. Fast variational static IR-drop analysis on the graphical processing unit. *In Proc. of International Symposium on Quality Electronic Design*. 1-6.

Vincenzi, A., Sridhar, A., Ruggiero, M., Atienza, D. 2011. Fast thermal simulation of 2D/3D integrated circuits exploiting neural networks and GPUs. *In Proc. of International Symposium on Low-Power Electronics and Design*. 151-156.

Vinco, S., Chatterjee. D., Bertacco, V., and Fummi, F. 2012. SAGA: SystemC acceleration on GPU architectures. *In Proc. of Design Automation Conference*. 115-120.

Vineet, V., and Narayanan, P. 2008. Cuda cuts: Fast graph cuts on the GPU. In *Proc. of Conference on Computer Vision and Pattern Recognition: Workshop on Visual Computer Vision on GPUs*. 1–8.

Vineet, V., Harish, P., Patidar, S., and Narayanan, P. J. 2009. Fast minimum spanning tree for large graphs on the GPU. *In Proceeding of High Performance Graphics*. 167-171.

Viswanathan, N., Pan, M., and Chu, C. 2007. Fastplace 3.0: a fast Multilevel quadratic placement algorithm with placement congestion control. *In Proc. of Asia South Pacific Design Automation Conference*. 135-140.

Volkov, V. and Demmel, J. 2008. Benchmarking GPUs to tune dense linear algebra. *In Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*. 1-11.

Volkov, V. 2010. Better performance at lower occupancy. *In Proc. of GPU Technology Conference*.

Wang, B., Zhu, Y., and Deng, Y. 2010. Distributed time, conservative parallel logic simulation on GPUs. *In Proc. of Design Automation Conference*.

Wang, L.-T., Wu, C.-W., and Wen, X. 2006. VLSI test principles and architectures: design for testability. Morgan Kaufmann. 1st edition. 761-766.

Waterman, M. S. 1995. Introduction to computational biology: maps, sequences and genomes. Chapman and Hall/CRC. 1st edition.

Wikipedia. 2012. List of Intel microprocessors. http://en.wikipedia.org/wiki/List_of_Intel_microprocessors.

Williamson, J., Lu, Y., Shang, L., Zhou, H., and Zeng, X. 2011. Parallel cross-layer optimization of high-level synthesis and physical design. *In Proc. of Design Automation Conferenc*e. 467-472.

Wittenbrink, C. M., Kilgariff, E. and Prabhu, A. 2011. Fermi GF100 GPU Architecture. *IEEE Micro*. 21(2):50-59.

Xiao, X., Aji, A. M., and Feng, W.-C. 2009. On the robust mapping of dynamic programming onto a graphics processing Unit. *In Proc. of International Conference on Parallel and Distributed Systems*. 26-33.

Xiao, X., and Feng, W.-C. 2010. Inter-block GPU communication via fast barrier synchronization. *In Proc. of IEEE International Symposium on Parallel and Distributed Processing*. 1-12.

Yang, Y., Xiang, P., Kong, J., and Zhou, H. 2010. A GPGPU compiler for memory optimization and parallelism management. *In Proc. of Conference on Programming Language Design and Implementation*. 86-97.

Yu, Z., Guan, X., Deng, Y. and Wang, Y. 2009. Riding bandwagon of parallel computing for nanoscale device simulation. *In Proc. of International Workshop on Quantum Systems and Semiconductor Devices: Analysis, Simulations, Applications*.

Zeng, Z., Ye, X., Feng, Z., and Li, P. 2010. Tradeoff analysis and optimization of power delivery networks with on-chip voltage regulation. *In Proc. of Design Automation Conference*. 831-836.

Zhang, J., et al. 2009. GPU-accelerated inverse lithography technique. *In Proc. of SPIE*. Vol. 7379.

Zhao, X., and Feng, Z. 2011. Fast multipole method on GPU: tackling 3-D capacitance extraction on massively parallel SIMD platforms. *In Proc. of Design Automation Conference*. 558-563.

Zhu, Y., Wang, B., Deng, Y. 2011. Massively parallel logic simulation with GPUs. *ACM Transactions on Design Automation of Electronic Systems*. 16(3).