Load Balancing and Dynamic Parallelism for Static and Morph Graph Algorithms

A Project Report
submitted in partial fulfilment of the
requirements for the Degree of

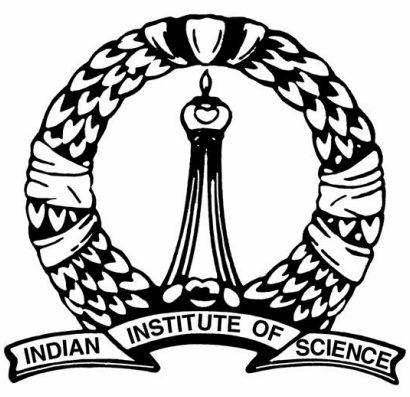
Master of Technology
in
Computational Science

by

Vivek K

under the guidance of

Dr. Sathish Vadhiyar



Super Computer Education and Research Centre
Indian Institute of Science
Bangalore – 560 012 (INDIA)

JUNE 2013

Dedicated to

My Dear Parents, Brothers and Friends

Acknowledgements

I would like to thank my project guide, Dr. Sathish Vadhiyar, for his guidance and encouragement throughout my work. Without his motivation this project work would not have been possible. I would like to thank Prof. Matthew Jacob, Dr. Venkatesh Babu, Dr. Murugesan Venkatapathi and all teachers who helped me in my project with their valuable suggestions and courses that I studied under them.

I would like to thank SERC for allowing me to use the system resources extensively for my research work. The course work in SERC was very helpful for my project. The system administrators in SERC always maintained a friendly countenance and were prompt in responding to the issues related to system resources. I would like to thank Rupesh Nasre (The University of Texas at Austin) and Jonathan Richard Shewchuk (University of California at Berkeley) for providing input data set for our experiments.

I would like to thank, all my MTech batch-mates, Calvin and Jaya Prakash who were always helpful during the hour of need. They made both the academic and the non-academic life enjoyable during my stay in Indian Institute of Science. I would also like to thank my family, friends. Special thanks to my lab mates - Sai Kiran, Anurag and Vasu for their constant support and encouragement.

Abstract

Modern GPUs are very powerful in accelerating regular applications. However, the performance of the irregular applications suffer mainly due to the load imbalance among the GPU threads. In this project, we consider two kinds of irregular applications, namely, static and morph graph algorithms. We first propose a novel node splitting strategy for load balancing in static graph algorithms. The performance of node splitting approach is based on optimum node splitting level. The best case node splitting for BFS algorithm improves performance by up to 55.8% for a real world web graph. We also propose a histogram based heuristic approach which automatically selects the node splitting level. We have also explored the dynamic parallelism offered by Kepler architectures as an alternative method of load balancing in both static and morph graph algorithms, and provide useful insights into the benefits of dynamic parallelism.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Graphics Processing Unit (GPU) has hundreds of SIMD cores which makes it a powerful accelerator for compute intensive tasks. Because of its high peak performance and cost effectiveness, GPU is being increasingly used for general purpose computing. GPU is well suited for regular applications that exhibit extensive data parallelism, where control flow and data access patterns are known at compile time. So, many efforts exist on fine tuning regular applications for GPUs [1, 2, 3].

In irregular applications, control flow and data access patterns are based on the run time data values. Even though irregular applications exhibit data parallelism, load incurred by threads are based on the data. This could lead to an imbalance in load because of which irregular applications are not well suited for GPU compared to regular applications. Hence, load balancing irregular applications is challenging.

Recently, the GPU community has devoted much attention to graph algorithms like Breadth-First Search (BFS) and Single-Source Shortest Paths (SSSP) [4, 5, 6, 7]. These are static graph algorithms which means the graph is not changed at run time . We believe that optimizations and load balancing strategies applied to these algorithms can be applied to other static irregular graph algorithms. On the other hand, Delaunay Mesh Refinement (DMR) [8, 9] is a morph algorithm, which means that the algorithm changes the graph by adding and deleting nodes and edges.

All these graph applications require load balancing to exploit the GPU to the full extent. Since the irregular applications' load are based on the run time data, dynamic load balancing is

necessary. In this work, we propose a novel load balancing strategy for static graph algorithms based on the concept of *node splitting*. In this approach, nodes which have an out degree greater than a threshold are split into lesser out degree nodes, thereby ensuring load balancing. The performance of node splitting approach is based on optimum node splitting level. We also propose a histogram based heuristic approach which automatically selects the node splitting level. Our load balancing strategy gives up to 55.8% improvement in performance for BFS and up to 50.8% performance improvement for SSSP.

NVIDIA has introduced dynamic parallelism in their latest GPU, Kepler. This feature can be availed for devices with the compute capability 3.5 and above. Using this feature, kernels can be launched within a kernel. So computational power can be allocated to regions of interest by invoking child kernels. However, this new feature comes with the overhead because of device runtime's execution tracking and management software. We have also explored the dynamic parallelism as an alternative method of load balancing in both static and morph graph algorithms, and provide useful insights into the benefits of dynamic parallelism.

The following are the contributions of this work.

- We propose a novel node splitting load balancing strategy for BFS and SSSP.
- For selecting node splitting level automatically, we propose histogram based heuristic approach.
- We also use dynamic parallelism to load balance BFS and SSSP, and compare the performance improvement with our proposed node splitting strategy.
- We observe performance improvement in DMR over probabilistic 3-phase conflict resolution technique by Nasre et al.[10], due to the effective use of atomic instructions along with the 3-phase technique.
- For load balancing the re-triangulation phase of DMR, we employ dynamic parallelism and work stealing and observe the impact in the performance.

The rest of the report is organized as follows. Chapter 2 briefly reviews the background. Chapter 3 discusses the related work. Chapter 4 and Chapter 5 describe the details of our

proposed node splitting strategy, histogram approach for determining node splitting level and optimizations applied to DMR. In Chapter 6, we present our experimental methodology and results. Chapter 7 concludes with the future work.

Chapter 2

Background

2.1 GPU Architecture and CUDA

After the introduction of NVIDIA's G80 unified graphics and compute architecture, GPUs have got tremendous attention among high performance computing community. NVIDIA has also introduced CUDA programming language which is a C++ extension, which makes the GPU programming easier than ever. CPU initiates the execution on GPU, the GPU being a slave device. GPU has a dedicated off chip DRAM called device memory connected with high speed bus with high latency. Since GPU cannot access host memory as fast as device memory, before proceeding with the execution on GPU, data must be transferred from host to device memory.

GPU consists tens of streaming multiprocessors (SM) which in turn consist of lot of CUDA cores. For example, kepler GK110 series GPUs have 13 new streaming multiprocessors (SMx), each having 192 cores[19]. CUDA runtime assigns one or more thread blocks to same SM, which depends upon the resource requirement by the thread blocks and hardware limitations of the SM. But SM will not share thread blocks. Warp is the basic scheduling unit of GPU, which consists of 32 threads. Threads in a warp execute the same instruction, any thread divergence leads to performance penalty. SMs are connected to the device memory with a high bandwidth bus. Memory coalescing improves the bus efficiency whereas, occupancy plays major role in hiding bus latency. GPU memory sub system is organized in a hierarchical way. All SMs share common L2 cache. The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations.

Threads within a block can communicate by shared memory or global memory. A new shuffle instruction has been introduced in kepler GPUs which allows threads within a warp to share data without the use of shared memory. Synchronization within a block can be achieved by high speed hardware synchronization primitive called __syncthreads(). Writes to global memory by a thread are not guaranteed to be visible to threads in other blocks. This can be solved using __threadfence(), it makes calling thread to wait until the writes are visible to other blocks.

2.2 Dynamic Parallelism

Without dynamic parallelism, it would have been a nightmare for the programmers to express algorithms which requires recursion and irregular loop structures[20]. Dynamic parallelism provides the opportunity to launch kernels from within a kernel, thereby reducing the transfer of execution control and data between host and device. Figure 2.1 illustrates implicit synchronization between parent and child kernels, forcing the parent to wait until all its children exit. Execution of a thread block is not considered complete until all launches by all threads in the block have completed. Like parent kernel, child kernel is also asynchronous. The child grid may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an explicit launch-synchronization point.

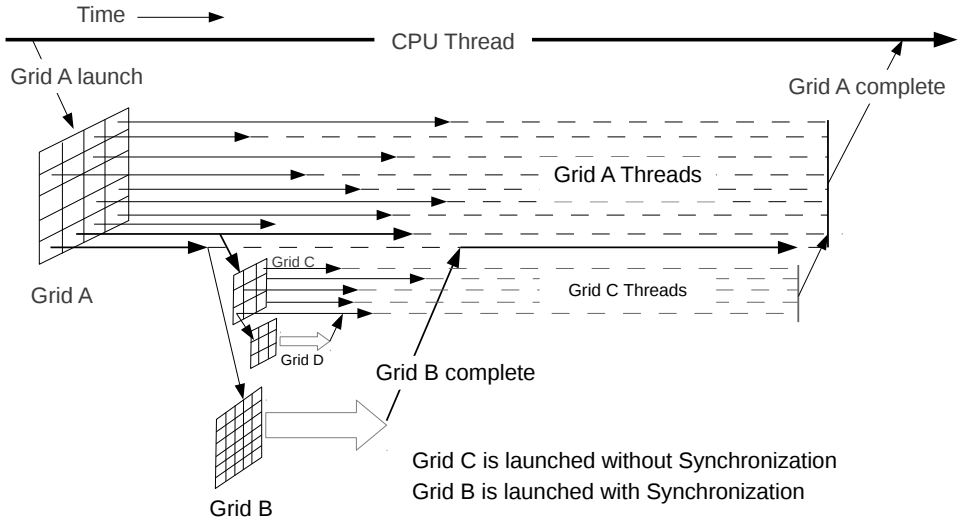


Figure 2.1: Dynamic Parallelism - Nested Kernel Launch

Chapter 3

Related Work

Pingali et al.[11] introduces a data-centric formulation of algorithms called the operator formulation. This formulation is the basis for tao-analysis which classifies the algorithms based on the operator into morph algorithms, local computation algorithms and reader algorithms. They also show that many irregular programs have amorphous data-parallelism. This has led to the development of Galois system that automatically executes "Galoized" serial C++ or Java code in parallel on shared-memory machines[12]. It works by exploiting amorphous data-parallelism, which is present even in irregular graph algorithms. The Galois system includes the Lonestar benchmark suite[13]. Recent efforts by Nasre et al.[14] have utilized the GPU for implementing few Lonestar benchmark algorithms and observed performance improvement over Galois multi-core system.

In [15], Nasre et al. answers when is it better to use data-driven versus topology-driven processing for implementing graph algorithms on GPUs, what are the tradeoffs and how to optimize such implementation. In the topology driven approach they explored strategies such as kernel unrolling, shared memory exploitation and memory layout optimizations. For BFS and SSSP it shown that topology-driven approach out performs data-driven approach.

Nasre et al.[10] have proposed efficient techniques to perform concurrent subgraph addition, subgraph deletion, conflict detection and several optimizations to improve the scalability of morph algorithms on GPU, and have observed performance improvement compared to multi-core implementation. For the DMR, they achieved speed up of 54-80$\times$ compared to serial version.

Nasre et al. in [16] have explored the benefits of avoiding atomic instructions on GPUs for irregular applications. They observed an improvement of 16% for SSSP, 11% for BFS and 29% for DMR over an atomic based implementation using Fermi GPU.

In all these works, there is no effort to balance the load among the GPU threads. In this project we perform load balancing to get a improved performance. In BFS and SSSP we use a *node splitting* strategy for load balancing. We use work stealing technique for load balancing in re-triangulation phase of DMR. We also use dynamic parallelism of Kepler GPUs to load balance these three applications. The selective use of atomic instructions for these applications provides opportunities to avoid under utilization of GPU resources and avoids redundant computations for DMR. Other works on static graph algorithms includes work by Merrill et al. [4], Luo et al. [5], Hong et al. [6], Harish et al. [7], and Hong et al. [17]. For DMR application, a refinement algorithm has been proposed by Navarro et al. [18].

Chapter 4

Load Balancing Static Graph Algorithms

Graphs can be represented using pointer based data structures. Compressed sparse graph representation is a popular choice, which is shown in Figure 4.1. We use fixed-point algorithm for BFS and SSSP where algorithm continuously update the node level in the case of BFS and distance in the case of SSSP until no more updates are made. The only difference in SSSP fixed point algorithm compared to that of BFS is that, in the former, edge weights are part of the input whereas, in the later edge weight is one.

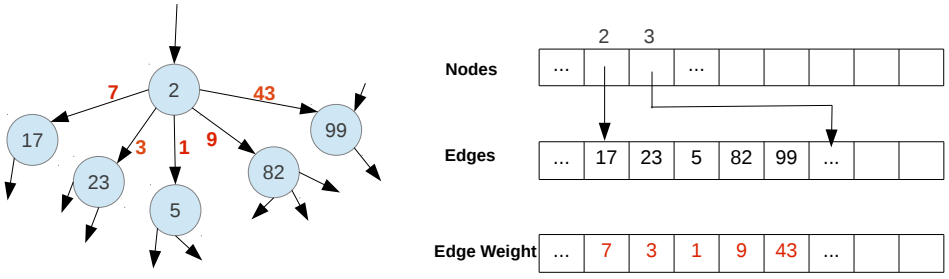


Figure 4.1: Compressed Sparse Graph Representation

4.1 Motivation

Figure 4.2 and Figure 4.3 shows the degree distribution of synthetically generated graph (R-MAT) and Stanford web graph respectively. From both these graphs, we can observe that significant amount of large out degree nodes are present. If one thread is assigned to relax the

out going edges of one node, then the threads assigned to nodes with high out degree will take a significantly longer time to finish computation. This causes the threads assigned to nodes with small out degree to have large waiting time. Since this causes an inefficient utilization of GPU resources, there is a need to balance the load among the threads.

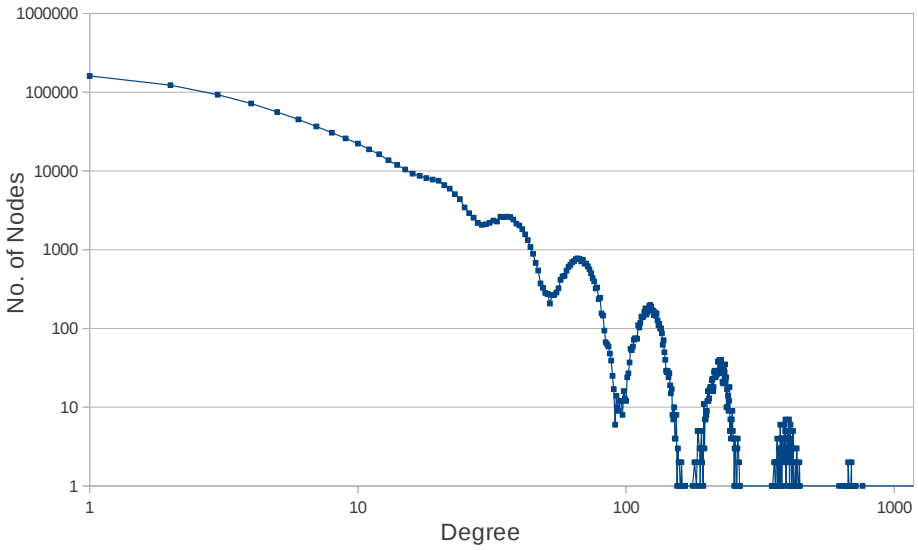


Figure 4.2: Degree Distribution of R-MAT graph

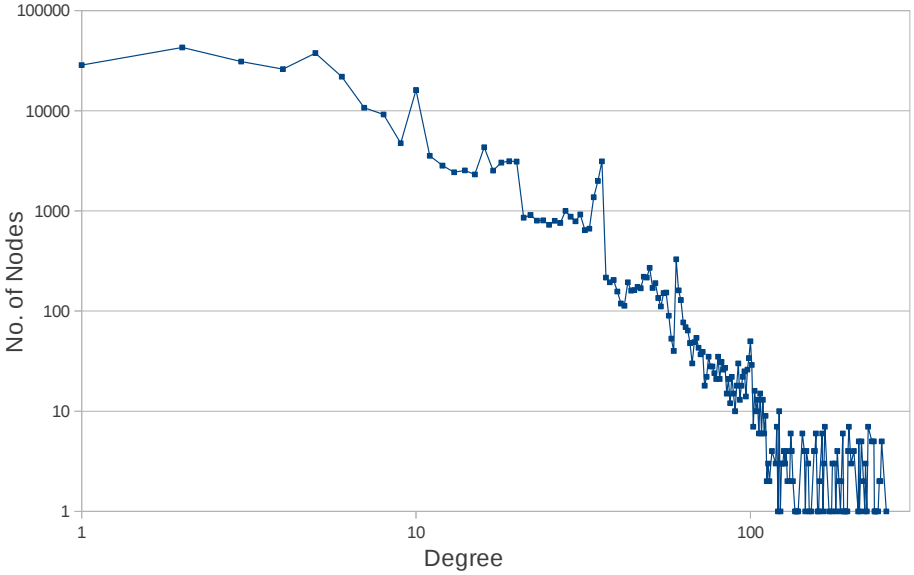


Figure 4.3: Degree Distribution of Stanford Web graph

4.2 Load Balancing Using Node Splitting

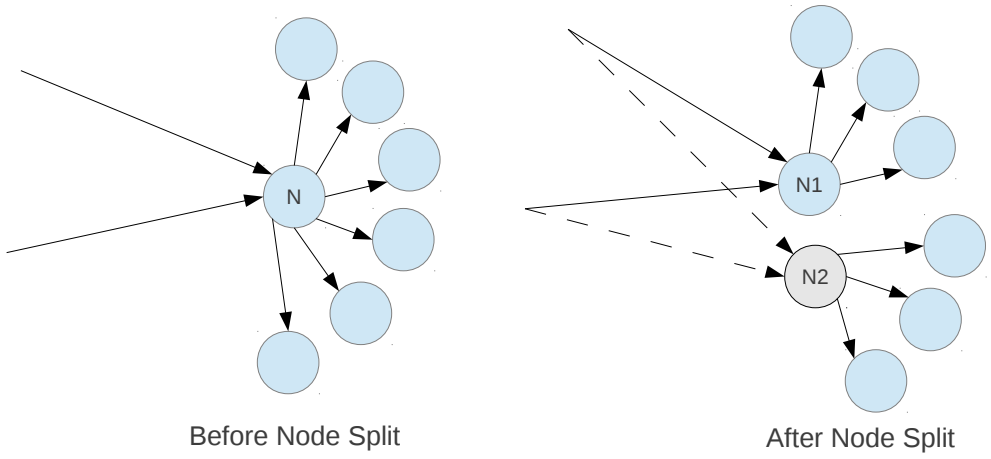


Figure 4.4: Node Splitting Strategy

This approach works based on the input parameter called maximum-out-degree-allowed (MDA). If a node's out degree is more than MDA, then that node will be split into $\lceil \frac{Outdegree}{MDA} \rceil$ nodes, each node with more or less equal out degree. For example, Figure 4.4 depicts our node splitting approach, where a node N split into two nodes N1 and N2 which share the edges of N. After node split N1 is called parent node, node N2 is child node. More than one child nodes are possible for any parent nodes. Inspite of adding an extra node, no new edges are added to the graph data structure. But the incoming edges for node N are present only on node N1 and the distance of N1 must get reflected on the node N2 also. To achieve this every node maintains a count of its children by which it can update the distance of the child nodes. This is represented in the Figure as dotted edges called pseudo edges.

Pseudo code for fixed point BFS implementation with node splitting approach is shown in Algorithm 1. Given an input graph, based on the MDA value it is load balanced using the node splitting strategy. The relax_kernel relaxes the out going edges of every node. Updating child nodes' distance in relax_kernel is necessary for the converge of the algorithm, but it is not guaranteed that all child nodes have the same distance as that of the parent nodes. So childNodeDistUpdate_kernel() is launched to update the distance of child nodes with that of the parent nodes. These two kernels are called until no more edges are relaxed.

main():
read input graph
nodeSplitFunction(MDA)
initialize distance vector
transfer graph and distance `// CPU -> GPU`
while *changed* **do**
    relax_kernel()
    childNodeDistUpdate_kernel()
    transfer changed `// GPU -> CPU`
end
transfer final distance `// GPU -> CPU`

relax_kernel():
foreach *node n in my nodes* **do**
    foreach *Edge e of n* **do**
        if *dist[e.node] > dist[n] + e.weight* **then**
            - dist[e.node] = dist[n] + e.weight `// relax`
            - Update e.node's children dist also `// pseudo edges`
            - mark changed as true
        end
    end
end

Algorithm 1: Pseudo code for fixed point BFS with node split

4.2.1 Automatic Determination of Node Splitting Level

In our graph algorithms, the main concern is to avoid more work being assigned to a single thread which could result in an entire block waiting for a thread to complete. Choosing correct MDA is necessary to obtain maximum performance due to load balancing.

We propose an histogram-based approach for automatically finding the node splitting level. In our approach, first we find the maximum out degree of the graph, using which, we create ten buckets uniformly. Then these buckets are filled based on the node's out degree. If bucket B has the maximum number of nodes, MDA is chosen to be the degree of the largest out degree node that can fit into B. Therefore all nodes falling beyond bucket B are split and made to fall in buckets with range equal to or lower than B.

4.3 Load Balancing using Dynamic Parallelism

Dynamic parallelism allows the worker count to be increased dynamically based on the work size. Graph exploration algorithms like BFS and SSSP have got varying number of out degree nodes which is known only at run time. In these algorithms, work size is related to the out degree of the nodes. High out degree nodes (determined using predefined threshold parameter like MDA) can be processed by launching child kernel on GPUs which creates more threads. So, parent thread's work is shared by the threads in the child kernel.

4.4 Other Optimizations - Atomic Instructions for Edge Relaxation

In the fixed point BFS implementation, more than one thread may update the node distance. But there is no guarantee that minimum distance will be written at last. As a result, convergence rate is reduced. The convergence rate can be improved using atomicMin instruction while relaxing the edges. This instruction is available only for integer distance, so it can not be used for SSSP with real valued edge weights. When atomicMin is used, childNodeDistUpdate_kernel() need not be launched, because atomicMin will ensure consistent distance among parent and child nodes.

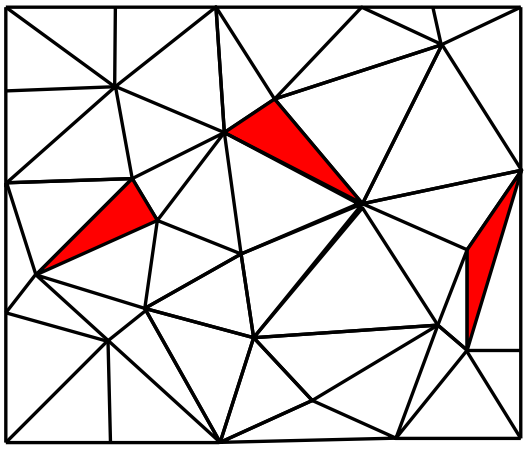
Chapter 5

Load Balancing Morph Graph Algorithms

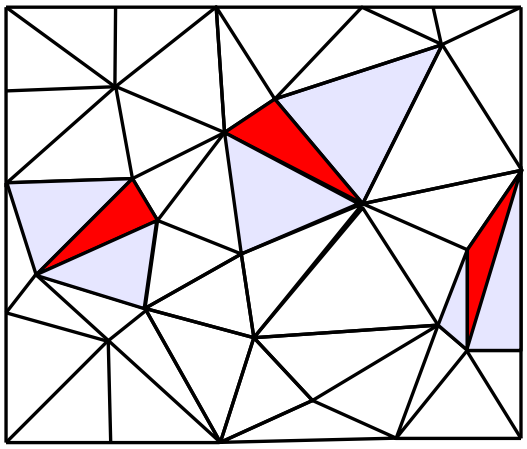
We use Delaunay Mesh Refinement as a case study for morph graph algorithms. Two dimensional DMR takes an input delaunay mesh which may contain bad triangles that do not meet the specified quality constraints, and produces a refined mesh by iteratively re-triangulating the affected portions of the mesh[21]. Quality constraint of the triangle is generally a minimum angle constraint. Ruppert's[9] iterative algorithm is used for refining the mesh. Following are the steps involved in DMR.

- First step is identifying bad triangles in the mesh as shown in Figure 5.1a.
- In the second step as shown in Figure 5.1b, cavities are formed around the bad triangles. Triangles whose circumcircle contains the circumcenter of the bad triangles are included in the respective bad triangles' cavity.
- Third step is deleting triangles in the cavities and adding new triangles whose vertex includes circumcenter of bad triangle, which is shown in Figure 5.1c.
- Some of the newly added triangles may not satisfy the quality constraints as shown in Figure 5.1d. Such triangles have to be marked as bad triangles, which can be refined in the subsequent iterations.

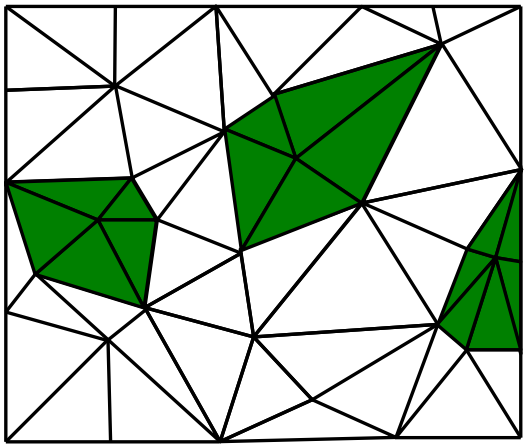
The algorithms seems not to have any parallelism. But intuitively it is clear that any non overlapping cavities can be processed independently. This kind of parallelism is called amorphous



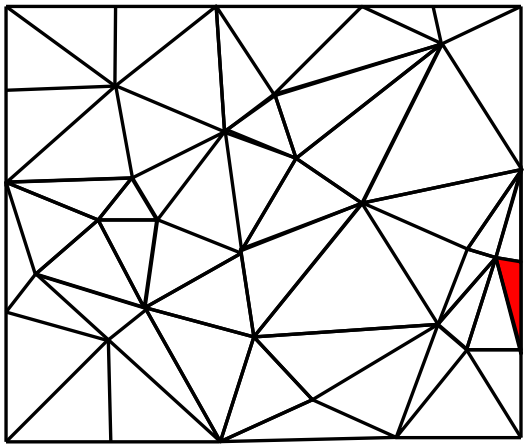
(a) Initial Bad Triangles



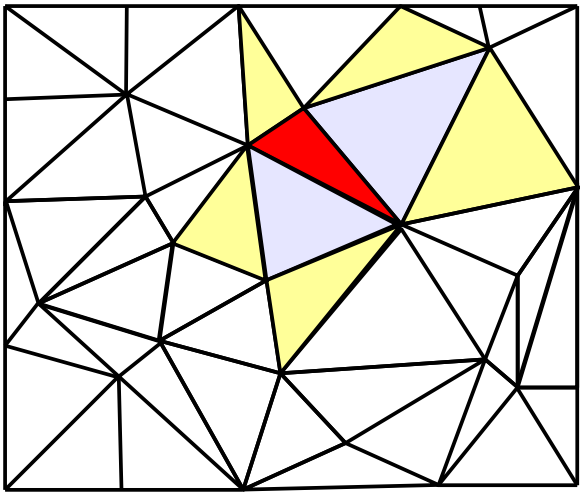
(b) Cavity Formulation



(c) Retriangulation



(d) New Bad Triangles



(e) Cavity formulation in parallel implementation

Figure 5.1: Steps in DMR

data parallelism[11, 10]. As shown in Figure 5.1e, in parallel implementation, in addition to including the triangles explained above, cavities should also include outer neighbor triangles (shaded in yellow) so as to ensure that the yellow triangles are not deleted due to other cavities. This maintains consistent neighbor relation between the triangles.

5.1 DMR on GPUs

Since DMR is a morph algorithm, it adds and deletes triangles at run time, necessitating lot of synchronizations. Identifying non overlapping cavities requires the use of atomic instructions. Nasre et al. [10] have efficiently implemented DMR on GPU by overcoming these challenges. The probabilistic 3-phase conflict resolution technique devised by the work avoids the use of atomic instructions to identify the non overlapping cavities in each iteration, aiding faster execution. In their work, since they launch a single kernel for all steps of DMR, they require a global barrier after each step. In their implementation for __global_sync(), they use an efficient atomic free global barrier proposed by Xiao and Feng [**?**].

5.2 Load Balancing using Work Stealing

In the re-triangulation phase, the number of new triangles added varies from 5 to 15 depending upon the input mesh. Also in this phase, not all the threads will be involved in re-triangulation which results in such threads being idle. Checking the quality constraint of newly added triangles is a compute intensive task. In this work stealing technique, we write all new triangles' id to a global memory and the total number of new triangles is calculated. Then each thread is assigned equal number of triangles to process.

5.3 Load Balancing using Dynamic Parallelism

After the addition of new triangles in the re-triangulation phase, using dynamic parallelism, a new kernel is launched with more number of threads so that the quality check of each new triangle is shared among these threads. For example, in the re-triangulation phase, after 15 new

triangles are added, instead of using one thread to handle the quality constraint computation, using dynamic parallelism we can launch a child kernel with 15 threads, one for each newly added triangle.

5.4 Other optimizations - Barrier Free Two Stage Implementation of DMR

In the work by Nasre et al.[10], global barrier is an essential part. This barrier is used in the 3-phase marking scheme of their implementation. For the correctness of the algorithm, it is essential that marking, re-marking and re-triangulation phases, all part of the same kernel, must execute one after the other. Therefore global barrier is used to achieve this objective. Global barrier can be achieved only among the active thread blocks. In their implementation, threads which do not get the bad triangles to form the cavity will be idle in subsequent phases while the other threads form cavities and re-triangulate formed cavities Although 3-phase marking scheme is efficient compared to atomic marking, in some cases even though more than one cavities can be processed in parallel, this marking scheme will not allow any threads to process the cavities. For example, if three cavities, C1, C2 and C3, overlap in two triangles, t1 and t2, the end of marking phase may result in t1 and t2 being owned by C3. After the re-marking phase, it is possible that t1 is owned by C1 and t2 is owned by C2. Now we see that no single cavity owns all the triangles in it. So all the three cavities will not be processed in this iteration. If no change occurs in the mesh due to this reason, the next iteration is executed using a single thread which could under-utilize the GPU.

Algorithm 2 shows our barrier free two stage implementation of DMR. Both the stages involve formulation of cavities, identifying non-overlapping cavities and re-triangulation of cavities. We do not use global barrier since synchronization is inherent between two kernel launches. First stage uses 3-phase marking scheme [10] and the second stage uses atomic marking scheme for identifying non overlapping cavities.

In the first stage, markCavity_kernel() does the formulation of cavities and marks each cavity with an unique id. All the kernels other than markCavity_kernel() present in both the

stages are launched with the number of threads equal to number of cavities. Each thread of re-markCavity_kernel() is responsible for one cavity, it reads the owner ids of the triangles in the cavity and if the owner id is greater than the cavity's id, it re-marks the cavity with its unique id. In retriangulation_kernel(), threads check if all the triangles of the respective cavities' are marked with cavities' unique id, thereby ensuring its cavity is not overlapping with any other cavity. Then it does the re-triangulation for its cavity.

A cavity may contain more than one bad triangle, but it is created around one bad triangle, we refer to this bad triangle as *root bad triangle*. *Alive cavities* are those whose root bad triangle is not deleted. If any of the triangles of a cavity are deleted, then the cavity is refered to as *damaged cavity*.

In the first stage, most of the cavities are processed, the second stage processes the remaining cavities. In the second stage, checkCavity_kernel() rebuilds damaged alive cavities. Undamaged cavities are reused. Then markCavityAtomic_kernel() is launched, which uses atomic instructions to mark the owner id of the triangles of a cavity. Since, only few non processed cavities would exist in the second stage, the use of atomic instructions will not affect the performance much. Again retriangulation_kernel() is launched, which re-triangulates the cavities.

main():
read input mesh
transfer initial mesh to GPU
initialize_kernel()
while *changed* **do**
    `// first stage`
    markCavity_kernel()
    re-markCavity_kernel()
    retriangulation_kernel()
    `// second stage`
    while *there exists a non-processed cavity* **do**
        checkCavity_kernel()
        markCavityAtomic_kernel()
        retriangulation_kernel()
    end
    transfer changed
end

markCavity_kernel():
foreach *triangle t in threadwork* **do**
    if *t is bad and not deleted* **then**
        - create and expand cavity around t
        - mark the cavity as non-processed
        - mark all triangles in the cavity with my thread id
    end
end
re-markCavity_kernel():
foreach *Cavity cav in Cavities* **do**
    foreach *triangle t in cav* **do**
        if *t is marked with id greater than my id* **then**
            mark t with my id
        end
    end
end
retriangulation_kernel():
foreach *Non processed Cavity cav in Cavities* **do**
    if *all triangles in the cav marked with my id* **then**
        - mark cav as processed
        - mark triangles in the cav as deleted, add new triangles and update the neighbors
        - mark the newly added triangles as bad if it is so
        - set changed as true
    end
end
checkCavity_kernel():
foreach *Non processed Cavity cav in Cavities* **do**
    if *cav is not alive* **then**
        mark cav as processed
    else
        if cav is damaged then cav is rebuild.
    end
end

Algorithm 2: Our GPU implementation of DMR

Chapter 6

Experiments and Results

6.1 Experimental Setup

In our experiments, we compare results with the baseline GPU implementation. Experiments have been done using both Fermi and Kepler GPUs. Fermi based GPU is a 1.15 GHz Tesla C2070, with 14 SMs each having 32 CUDA cores (448 CUDA cores totally) with 6 GB of main memory, 768 KB of L2 cache and 32 KB of register per SM. Kepler based GPU is a 0.71 GHz Tesla K20c, with 13 SMs each having 192 CUDA cores (2496 CUDA cores totally) with 5 GB of main memory, 1 MB of L2 cache and 64 KB of register per SM. Both GPUs have configurable 64 KB of fast memory per SM that is split between the L1 data cache and shared memory. The programs have been compiled using nvcc version 5.0 with optimization flag -O3 , with -arch=sm_20 flag on Fermi and -arch=sm_35 flag on Kepler. For Kepler, dynamic parallelism enabled program, additional flags -rdc=true and -lcudadevrt are also included.

For experimenting BFS and SSSP we have used both synthetically generated graph (R-MAT) and real word Stanford web graph. R-MAT can be generated using [22]. In Stanford web graph, nodes represent pages from Stanford University and directed edges represent hyperlinks between them[23]. R-MAT has 1 M nodes and 8 M edges. Stanford web graph has 0.2 M nodes and 2 M edges.

Input meshes for DMR is generated using Triangle[24]. Three input mesh 0.5 M, 1.1 M, 5.6 M are generated, each having more than 50% bad triangles initially.

6.2 Results for Static Graph Algorithms

6.2.1 Node splitting

Figure 6.1 shows the degree distribution of Standford web graph after node splitting with Maximum Degree Allowed (MDA) chosen as 8. Comparing Figure 6.1 with Figure 4.3, we find that the nodes and the GPU threads are highly load imbalanced in terms of the number of out degrees to be processed.

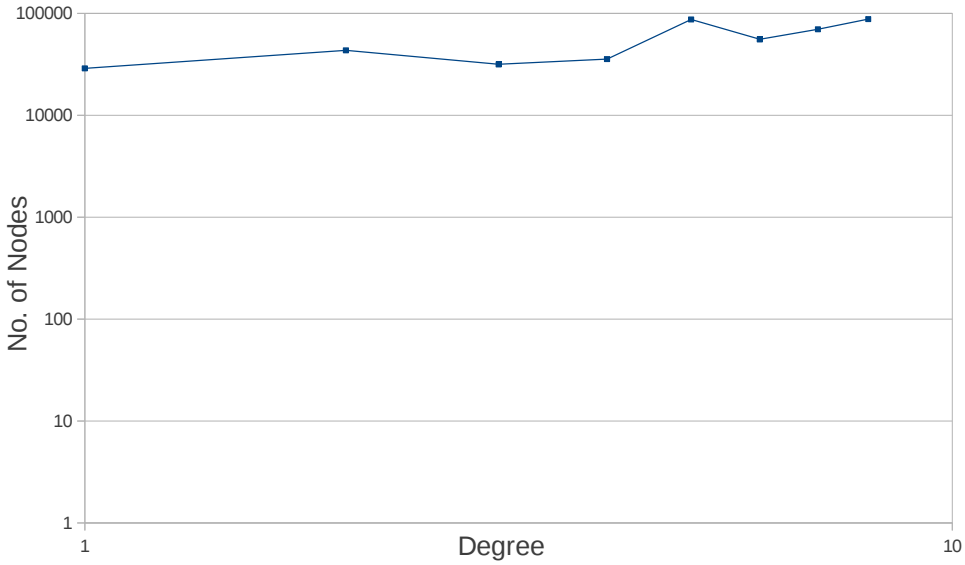
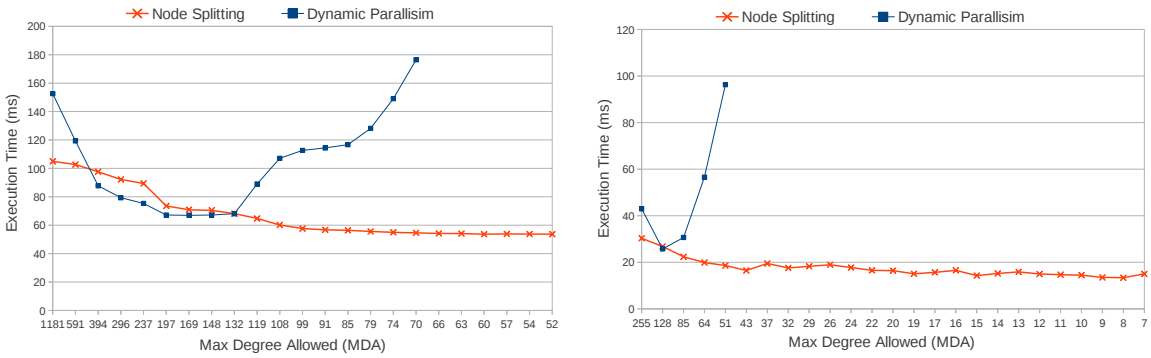


Figure 6.1: Degree Distribution of web graph after node splitting (MDA:8)

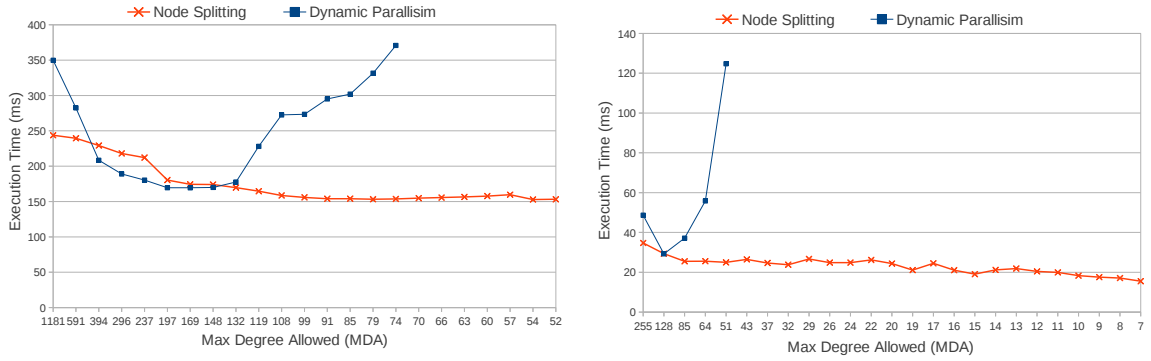
Figure 6.2 and Figure 6.3 shows the performance impact due to node splitting on Kepler GPU. In all the execution times shown for node splitting approach, kernel time for updating child node distance is also included, which is minimal for our input graphs. Maximum out degree of R-MAT graph is 1181 and that of Stanford web graph is 255. If these values are chosen as MDA, node splitting method executes baseline GPU implementation for both BFS and SSSP. For both BFS and SSSP, as MDA value decreases, we can observe that execution time decreases. Similar trend can be observed for Fermi GPU which is shown in Figure 6.4. Beyond certain MDA value, performance starts decreasing slowly due to excessive node splitting wherein threads

will be involved in updating distance for numerous child nodes. For R-MAT graph we get an improvement of 48.78% for BFS on Kepler, 37.3% for SSSP on Kepler, 24.8% for BFS on Fermi and 14.1% for SSSP on Fermi over baseline GPU implementation.



(a) R-MAT graph (b) Stanford web graph

Figure 6.2: BFS on Kepler

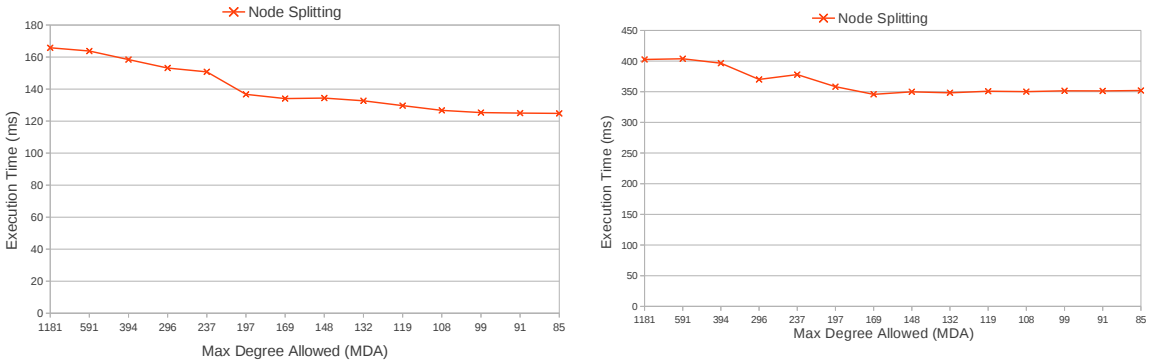


(a) R-MAT graph (b) Stanford web graph

Figure 6.3: SSSP on Kepler

From Figure 6.2 and Figure 6.3, we can observe that the BFS kernel executed faster than the SSSP kernel. This is because, in BFS all edge weights have the value one, so all nodes will reach the final distance faster than SSSP. Since Stanford web graph is smaller graph in terms of number of nodes and edges, it is processed faster than R-MAT graph. Because the computation power of Kepler GPUs is twice as much as Fermi GPUs, execution time of BFS and SSSP (Figure 6.2a and Figure 6.3a) is less on Kepler GPU when compared to Fermi GPU (Figure 6.4) for R-MAT

graph. The optimum node splitting point depends on the input graph.



(a) BFS (b) SSSP

Figure 6.4: BFS and SSSP on Fermi for R-MAT graph

Figure 6.5 shows the block level execution time on Fermi for the first iteration of BFS application using R-MAT graph. Each bar in the graph corresponds to the timeline of one SM and each block within it represents the execution time for a block assigned to the SM. From the figure, we can observe that in base implementation, the execution time for blocks vary indicating load imbalance whereas, in node splitting approach almost all the blocks take the same amount of time. Table 6.1 shows the performance improvement due to node splitting on Kepler. Node splitting results in upto 55.88% improvement over baseline method.

Graph	Graph Algorithm	Baseline execution time(ms)	Max. Degree Allowed	Extra Nodes Introduced	Node Spitting	
					Execution Time(ms)	% improvement
R-MAT	BFS	104.96	54	29604	53.76	48.78%
	SSSP	243.81			152.88	37.30%
Stanford Web Graph	BFS	30.29	8	158002	13.36	55.88%
	SSSP	34.71			17.09	50.75%

Table 6.1: Performance of node splitting on Kepler

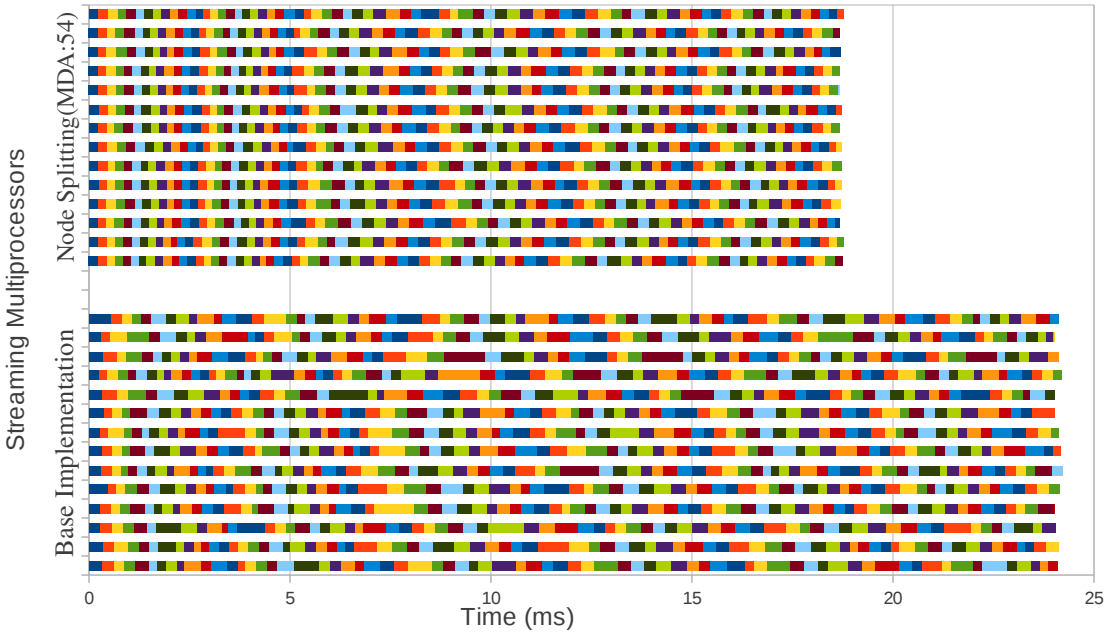


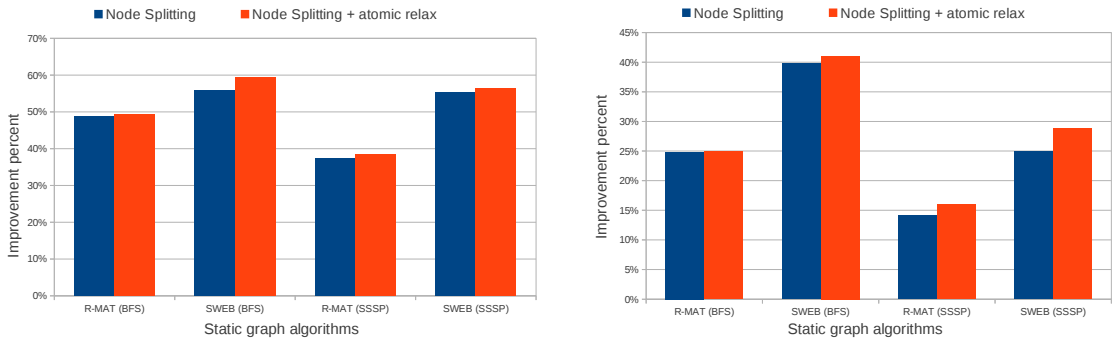
Figure 6.5: Block level BFS execution time for R-MAT graph on Fermi

6.2.2 Automatic Determination of Node Splitting Level

Here, we evaluate our histogram-based approach for automatic determination of node splitting level, described in Chapter 4.2.1. We have used ten buckets for our histogram heuristic. For R-MAT graph, the MDA value obtained using this approach is 118, which resulted in 38.3% improvement over baseline implementation. For Stanford web graph the value obtained is 25, for which 44.8% improvement over baseline implementation was obtained on Kepler. From the node distribution graphs (Figure 4.2 and Figure 4.3), we can observe that large number of nodes have very less out degree values. So in the histogram, bucket zero will have the largest count. Since we use ten buckets, for our input graphs, the histogram approach will result in a MDA value which is 10% of the maximum out degree of the graph as most of the nodes fall in bucket zero. For any other input graph the MDA value will change depending upon the most populated bucket.

6.2.3 Effect of using atomic instructions

Figure 6.6 shows the improvement obtained by using atomic operation for relaxing edges on both Fermi and Kepler GPUs. Atomic relax ensures the distance consistency among parent and child nodes. Since this operation ensures that parent and all its child nodes receive the minimum distance, convergence rate is faster than atomic-free version. The atomic relaxing gives an additional improvement of 2-6% over node splitting.



(a) Kepler (b) Fermi

Figure 6.6: Effect of using atomic instructions on static graph algorithms

6.2.4 Dynamic parallelism

Figures 6.2 and 6.3 also show the results for dynamic parallelism for BFS and SSSP on Kepler. An interesting observation corresponds to when MDA is set to maximum out degree. In this case, dynamic parallelism enabled kernel does not launch any child kernels. However, we find that for this case the execution time with dynamic parallelism (data point in blue curve) is higher than the execution time of the baseline implementation (data point in red curve). This overhead arises from the device runtime's execution tracking and management software of Kepler when dynamic parallelism is enabled. The execution time decreases as MDA decreases upto some point, after which it starts increasing. The increase in execution time is due to numerous child kernel launches for lower MDA values. We can observe that dynamic parallelism gives better performance initially compared to node splitting approach for same MDA, but node splitting ultimately wins for lesser MDA. It can be seen that the best MDA value for node splitting

doesn't result in best performance for dynamic parallelism. The performance improvement using dynamic parallelism is shown in Table 6.2. Dynamic parallelism results in improvement upto 36.19%.

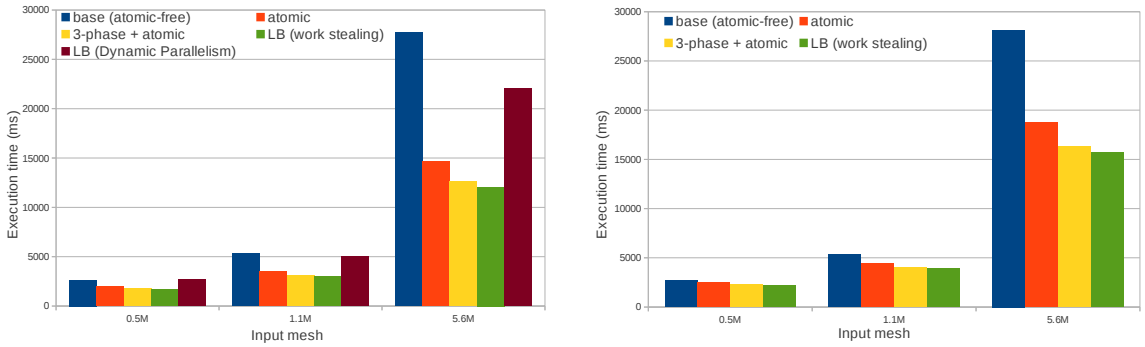
Graph	Graph Algorithm	Baseline execution time(ms)	Max. Degree Allowed	Dynamic Parallelism	
				Execution Time(ms)	% improvement
R-MAT	BFS	104.96	169	66.98	36.19%
	SSSP	243.81		169.44	30.50%
Stanford Web Graph	BFS	30.29	128	25.71	15.11%
	SSSP	34.71		29.24	15.76%

Table 6.2: Performance of dynamic parallelism on Kepler

6.3 Results for Morph Graph Algorithms

6.3.1 Optimization

Figure 6.7 shows the performance improvement due to various optimizations for DMR application experimented on both Fermi and Kepler. Here, base implementation refers to Nasre et al.[10] implementation of DMR which is atomic free implementation. The bar labelled *atomic* refers to our implementation of DMR which uses atomic instructions in both the stages (Algorithm 2) of identifying non overlapping cavities. The bar *atomic+3-phase* shows the execution time for our implementation using atomic free marking for the first stage and atomic marking for the second stage. *LB (work stealing)* adds load balancing to the previous optimization.



(a) Kepler (b) Fermi

Figure 6.7: Various optimizations of DMR on Kepler and Fermi GPUs

From Figure 6.8, we can observe that our implementation processes more triangles than base implementation as the computation progresses. This is because our implementation attempts to re-use the unprocessed cavities of the first stage in the second stage, so additional triangles are processed as base implementation uses only one stage. Since time taken for cavity formation and marking is saved for the unprocessed cavities in the second stage, we get better performance than base implementation.

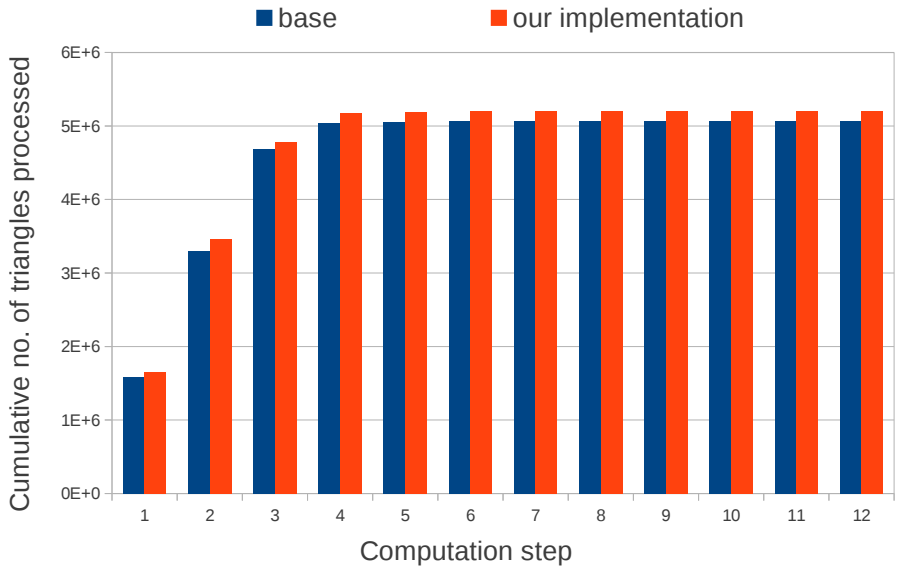


Figure 6.8: DMR with input mesh containing 1.1 M triangles on Kepler

Table 6.3 shows the execution time of DMR application using each of the optimization techniques. With all our optimizations applied, we get improvement of 56.51% on Kepler and 44.17% on Fermi for a 5.6 M triangles input mesh.

Optimization	Execution Time (ms)	
	Kepler	Fermi
base	27737.54	28144.12
only atomic	14661.88	18737.62
3-phase marking + atomic	12660.46	16291.16
3-phase marking + atomic with LB	12064.17	15713.53

Table 6.3: DMR results for input mesh containing 5.6 M triangles

6.3.2 Dynamic parallelism

The impact in performance due to dynamic parallelism is shown in Figure 6.7. Even though newly added triangles in re-triangulation phase varies from 5 to 15 for our input meshes, dynamic parallelism does not improve the performance because of the absence of sufficient work to overcome the overhead introduced.

6.4 Discussion: Insights on Dynamic Parallelism

We have observed that dynamic parallelism is useful only when the work load on a particular thread is high enough, so that when it is shared among the threads of the child kernel, each child thread has sufficient work. Also launching too many child kernels has an adverse effect on performance. Hence we suggest that these two factors must be taken into account while using dynamic parallelism.

Chapter 7

Conclusions and Future Work

In this project, we have focused on accelerating challenging irregular static and morph graph algorithms. We have employed novel node splitting strategy for BFS and SSSP for load balancing. Our strategy results in an improvement of 48.8% and 55.8% for synthetically generated and real world web graphs respectively on Kepler GPU. We have also experimented with dynamic parallelism for load balancing static graph algorithms and have obtained performance improvement upto 36.19%. We have optimized existing DMR implementation by effectively utilizing atomic instructions and load balancing. We have analyzed the performance impact of using dynamic parallelism for load balancing DMR. In both these algorithms, we have used atomic instructions effectively and have observed the performance improvement.

In the future, we plan to develop strategies for irregular application on multi-CPU and multi-GPU systems. We plan to understand the behavior of other irregular applications in future.

Bibliography

[1] Y. Liu, E. Z. Zhang, and X. Shen, "A Cross-input Adaptive Framework for GPU Program Optimizations," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[2] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense Linear Algebra Solvers for Multicore with GPU Accelerators," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[3] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 31.

[4] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 117–128.

[5] L. Luo, M. Wong, and W.-m. Hwu, "An Effective GPU Implementation of Breadth-First Search," in *Proceedings of the 47th design automation conference*. ACM, 2010, pp. 52–55.

[6] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-core CPU and GPU," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 78–88.

[7] P. Harish and P. Narayanan, "Accelerating Large graph Algorithms on the GPU Using CUDA," in *High Performance Computing–HiPC 2007*. Springer, 2007, pp. 197–208.

[8] L. P. Chew, "Guaranteed-quality Mesh Generation for Curved Surfaces," in *Proceedings of the ninth annual symposium on Computational geometry*. ACM, 1993, pp. 274–280.

[9] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-dimensional Mesh Generation," *J. Algorithms*, vol. 18, no. 3, pp. 548–585, 1995.

[10] R. Nasre, M. Burtscher, and K. Pingali, "Morph Algorithms on GPUs," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 147–156.

[11] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of Parallelism in Algorithms," *SIGPLAN Not.*, vol. 47, no. 6, pp. 12–25, Jun. 2011.

[12] Galois system. [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois

[13] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A Suite of Parallel Irregular Programs," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 65–76.

[14] LonestarGPU. [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

[15] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus Topology-driven Irregular Computations on GPUs," ser. IPDPS '13, 2013.

[16] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free Irregular Computations on GPUs," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6, 2013, pp. 96–107.

[17] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. ACM, 2011, pp. 267–276.

[18] C. A. Navarro, N. Hitschfeld-Kahler, and E. Scheihing, "A Parallel GPU-based Algorithm for Delaunay Edge-Flips," in *The 27th European Workshop on Computational Geometry, EuroCG*, vol. 11, 2011.

[19] NVIDIA Kepler GK110 Architecture Whitepaper. [Online]. Available: http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[20] CUDA Dynamic Parallelism Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf

[21] Ruppert's Delaunay Refinement Algorithm. [Online]. Available: https://www.cs.cmu.edu/~quake/tripaper/triangle3.html

[22] GTgraph : A suite of synthetic random graph generators. [Online]. Available: http://www.cse.psu.edu/~madduri/software/GTgraph/index.html

[23] Stanford Large Network Dataset Collection. [Online]. Available: http://snap.stanford.edu/data/index.html

[24] Triangle : A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. [Online]. Available: http://www.cs.cmu.edu/~quake/triangle.html