Walchand College Of Engineering, Sangli
(An Autonomous Institute)

B.TECH PROJECT REPORT

High Performance Content-Based Matching Using Multiple GPGPU and MPI GPGPU Approach

Submitted By:
Nitesh Sakle (2012BCS013)
Suyog Jain (2012BCS026)
Abhijeet Kulkarni (2012BCS031)

Project Guide:

Mrs. M. A. Shah

A project report submitted in fulfilment of the Mega Project for the degree of Bachelor of Technology

Department of Computer Science and Engineering

May 2016

Walchand College Of Engineering, Sangli
(An Autonomous Institute)

CERTIFICATE

This is to certify that, The Project Report entitled,

High Performance Content-Based Matching Using Multiple GPGPU and MPI GPGPU Approach

Submitted By:
Nitesh Sakle (2012BCS013)
Suyog Jain (2012BCS026)
Abhijeet Kulkarni (2012BCS031)

Is a bonafide record of their own work performed out by them in fulfillment of the final year B.Tech project in Computer Science and Engineering as specified in the curriculum prescribed by Walchand College of Engineering, Sangli.

Mrs. M. A. Shah
(Project Guide)

Dr. B. F. Momin
(Head Of Department)

Declaration

We, the undersigned, hereby declare that the project report entitled,

High Performance Content-Based Matching Using Multiple GPGPU and MPI GPGPU Approach

written and submitted by us to Department of Computer Science and Engineering, Walchand College of Engineering, Sangli as a complete fulfillment for Mega Project under the guidance of **Mrs. M. A. Shah** is our sincere work. The empirical results in this project report are based on the data collected by us. We understand that any type of copying is liable to be punished as the authorities deem fit.

Date :

Place : Walchand College Of Engineering, Sangli.

Nitesh Sakle (2012BCS013)
Suyog Jain (2012BCS026)
Abhijeet Kulkarni (2012BCS031)

Acknowledgement

We are pleased to present this project report entitled " High Performance Content-Based Matching Using Multiple GPGPU and MPI GPGPU Approach" to our college as a part of academic activity. We wish to take this opportunity to express our deep gratitude to all the people who have extended their co-operation in various ways during our project work. It is our pleasure to acknowledge the help of all those individuals. We are very thankful to our project guide **Mrs. M. A. Shah**. With her guidance, co-operation and encouragement we had learn many new things during our project tenure. We would like to specially thank **Dr. B.F. Momin**, HOD, Computer Science and Engineering Department, for his continuous encouragement and valuable guidance in bringing shape to this dissertation. We specially thank **Dr. G.V. Parishwad**, Director of Walchand College of Engineering, Sangli, for his support and encouragement. We are very thankful to all our colleagues and those who helped us directly or indirectly throughout this project work. We are thankful to college authorities. It won't be fair if we fail to acknowledge our family members, because of whom we are here. Their enthusiasm and backing have really boosted our confidence which has enabled us to complete our project work.

Nitesh Sakle (2012BCS013)
Suyog Jain (2012BCS026)
Abhijeet Kulkarni (2012BCS031)

Abstract

Matching incoming event notifications against received subscriptions is a fundamental part of every publisher-subscribe (pub-sub) infrastructure. In the past, several sequential and very few parallel algorithms have been proposed for efficient content-based event matching. Many-core General Purpose Graphics Processing Units (GPGPUs) have become an important computing platform in many scientific fields due to the high peak performance, cost effectiveness, and the availability of user-friendly programming environments, e.g., NVIDIA CUDA and ATI Stream. Content based matching approach using a single GPU have memory limitations for large, high dimensional, millions of subscriptions.

Our aim is to effectively exploit multiple GPGPUs concurrently, which are commonly available in many modern systems for increasing throughput and minimizing processing time. In this paper, we describe the Multi-GPGPU CUDA content matching algorithm. The algorithm is evaluated and tested against CCM. On multi-GPGPU systems, our solution achieves near-linear speedup, and significant performance improvement in terms of matching time and throughput. The multi-GPGPU algorithm achieves 1.6X higher throughput compared to the CCM and 40% achievement in reducing matching time.

Our objective to achieve more parallelism was also implemented at coarse grain level using a cluster of GPGPU using MPI programming which resulted in a performance increase upto 2x using four GPGPU considering the communication overhead.

Contents

List of Figures

Abbreviations

GPGPU	General Purpose Graphics Processing Unit
CCS	Cuda Computing System

Chapter 1
Introduction

In the near future, the majority of human information will be in the Web. The searching, querying and retrieving information approach, while very efficient for static information, does not integrate well with the dynamic aspect of the Web. To address this issue, notification systems emerged, being able to capture the dynamic aspect of the Web by notifying users of interesting events. These may vary from stock markets updates, weather reports, availability of auction items etc. In this context, a notification system was proposed called Le Subscribe, upon which the Event Notification system we present in this work was based. The Event Notification system is able to cope with high rate of events, large number of user demands (post subscription, remove subscription) and efficiently notify users with very short delay. Moreover it provides a simple and comprehensive API enabling easy integration of existing Web applications with an event notification mechanism.

Publisher-Subscriber System

Publish-subscribe (pub-sub) is an important paradigm for asynchronous communication between entities in a distributed network. In the pub-sub model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called filtering.

Content-based pub-sub is a more general and powerful paradigm, in which subscribers have the added flexibility of choosing filtering criteria along multiple dimensions, using thresholds and conditions on the contents of the message, rather than being restricted to (or even requiring) pre-defined subject fields. Content-based pub-sub applications present a unique challenge for efficient matching of events to subscriptions. In a large-scale pub-sub system, it is possible that there are millions of subscriptions maintained by brokers and millions of events to be matched every second. Therefore, to improve the matching speed as well as the throughput of brokers is of great importance to large-scale content-based publish/subscribe systems. Different techniques have been employed to improve matching efficiencies in recent years.

Despite their differences, they have a key aspect in common: they were all designed to run on conventional, sequential hardware. If this was reasonable years ago, when parallel hardware was the exception, today this is no more the case. Modern CPUs integrate multiple cores and more will be available in the immediate future. Moreover, modern General Purpose Graphical Processing Units (GPGPUs) integrate hundreds of cores, suitable for general purpose (not only graphic) processing.

Content vs. Subject-Based Systems

We can distinguish two kinds of pub/sub systems: subject-based and content-based. Subject-based systems, classify the events in groups or subjects and can only be filtered according to their group. A subject-based system would assign a group for each category. A publisher must assign each event with a group. Users can subscribe to groups, and be notified if an event belongs to groups of interest. For example, a subscriber can subscribe to be notified if a new auction item of type car is being put up. Content-based systems are the emerging type of pub/sub systems, where events are evaluated according to attribute-value pairs, against subscriptions posted by subscribers. This way, subscribers can have more specific subscriptions. Example of such subscription is an auction item of type car and price lower than 1300.

Comparing subject-based and content-based systems, the latter offer more subscription expressiveness, which of course has an impact in the complexity of the matching process. This complexity combined with a high event rate can severely degrade the matching efficiency. So systems devoted to handle high rate of events and a large number of subscribers have to face a trade of between the subscription language sophistication and matching efficiency.

Chapter 2

Literature Survey

Pub/Sub system with sequential matching algorithm

The last decade saw the development of a large number of content-based publish-subscribe systems [9], [10], [11], [12], first exploiting a centralized dispatcher, then moving to distributed solutions for improved scalability. Two main categories of matching algorithms have been proposed: counting-based [13, 14, 15] and tree-based [16, 17, 18] approaches. These approaches can further be classified as either key-based, in which for each expression a set of predicates are chosen as identifier [12], or as non-key based [13,15,17]. Counting-based methods aim to minimize the number of predicate evaluations by constructing an inverted index over all unique predicates. The two most efficient counting-based algorithms are Propagation [14], a key-based method, and the k-index [15], a non-key-based method. Likewise, tree-based methods are designed to reduce predicate evaluations and to recursively divide the search space by eliminating subscriptions on encountering unsatisfied predicates. The most prominent tree-based method, Gryphon, is a static, non-key based algorithm [16]. BE-Tree [20] is a novel tree-based approach, which also employs keys, that outperform existing work [13,16, 14,15]. The latest advancement of counting-based algorithms is k-index [15], which gracefully scales to thousands of dimensions and supports equality predicates ($\in$) and non-equality predicates ($\notin$). K-index partition subscriptions, based on their number of predicates to prune subscriptions with too few matching predicates; however, k-index is static and does not support dynamic insertion and deletion. BE-Tree is distinguished from k-index is that BE-Tree is fully dynamic, naturally supports richer predicate operators (e.g. range operators), and adapts to workload changes. BE-Tree, however, is limited to attributes whose values are discrete and for which the range in discrete attribute values is pre-specified. So, BE-tree is unable to cope with real-valued attributes, string-valued attributes, and discrete-valued attributes with unknown range. Additionally, BE-tree [20,21] employs a clustering policy that is ineffective when many subscriptions have a range predicate such as $low \leq a_i \leq high$, where a_i is an attribute and the clustering criterion *p* that is used lies between *low* and *high*. Here p is the clustering technique used to collect the subscriptions into the same cluster. In this case, all such subscriptions fall into the same cluster and event processing is considerably slowed. PUBSUB [19], which is a heterogeneous system, offers a variety of data structures to keep track of the buckets in an attribute structure enabling the user to select data structures best suited for each attribute. Because of PUBSUB's heterogeneity in data structures for each attribute, PUBSUB permits all attribute data types.

Several algorithms have been proposed for efficient content-based event matching. While they differ in most aspects, they have in common the fact of being conceived to run on conventional, sequential hardware.

High performance pub/sub systems

Here we only review work related to parallel and high performance event processing in publish subscribe systems. The idea of parallel matching has been recently addressed in a few works. In [2], the authors exploit multi core CPUs both to speed up the processing of a single event and to parallelize the processing of different events using threads. However, the processing delays and throughput appearing in the paper seem to be much worse than those obtained by our Multi GPGPU approach, due to use of limited threads available for processing. Other works investigated how to parallelize matching uses ad-hoc (FPGA) hardware [7]. The author in [1] described a new publish-subscribe content-based matching algorithm designed to run efficiently both on multi core CPUs and CUDA GPGPUs. The algorithm takes a substantial amount of time for processing large numbers of subscriptions, filters, interfaces. The efforts are taken to deploy pub sub system on storm [4] architecture for fast matching using local as well as distributed cluster. Resource utilization should be appropriate for better performance. Storm pub-sub system approximately produces 2200 event/s on distributed cluster. In compared to our multi GPGPU approach this throughput is low. Pub sub is experimented on a cluster with up to 384 cores indicate that Stream Hub [3] is able to register 150 K subscriptions per second and filter next to 2 K publications against 100 K stored subscriptions, resulting in nearly 400 K notifications sent per second. Deadline aware algorithm [6] is designed to maintain Quality of Service in pub/sub is modified, originally mentioned in [5]. In modified smart dispatch algorithm, number of failures decreases with the increase in number of cores. This paper uses the approach of parallelism using multithreading, so limited scalability is achieved. Our aim is to make high performance and scalable publish subscribe system by processing events in parallel.

Our objectives are as follows:

1. Find out the scope of parallelism in the matching process.
2. Design and development of parallel, scalable content matching algorithm.
3. Evaluation of parallel algorithm on different parallel platforms.
4. Performance analysis and suggestion of the suitable framework for high performance and scalable publish subscribe system.

We designed a new publish-subscribe content-based matching algorithm designed to run efficiently both on multi core CPUs and CUDA GPGPUs. The algorithm takes a substantial amount of time for processing large values of subscriptions, filters, interfaces. The efforts are taken to deploy pub sub system on storm architecture for fast matching using local as well as distributed cluster. Storm pub-sub system approximately produces 2200 event/s on distributed cluster. Compared to multi GPGPU approach this throughput is low. Pub sub is experimented on a cluster with up to 384 cores indicate that Stream Hub is able to register 150 K subscriptions per second and filter next to 2 K publications against 100 K stored subscriptions, resulting in nearly 400 K notifications sent per second. Deadline aware algorithm to maintain Quality of Service in pub-sub is modified originally mentioned in. In modified smart dispatch algorithm, the number of failures decreases with the increase in number of cores

Looking to earlier solutions, our aim is to make high performance and scalable publish subscribe system using multiple GPGPU .The main contributions of this work are:

1. The development, analysis and experimental evaluation of a novel, efficient multi-GPGPU and MPI-GPGPU algorithm that enables matching of events with a large number of subscriptions, executed in multiple GPGPU.
2. Design of decomposition strategies for distribution of events and subscriptions across multiple GPGPU.
3. Analysis of all the approaches and comparison of performance efficiency.

Cuda Content-Based Matcher

The CUDA Content-based Matcher (CCM) algorithm we designed is composed of two phases: a constraint selection phase and a constraint evaluation and counting phase. When an event enters the engine, the first phase is used to select, for each attribute a, of the event, all the constraints having the same name as a. These constraints are evaluated in the second phase, using the value of a. In particular, we keep a counter of satisfied constraints for each filter stored by the system: when a constraint c is satisfied, we increase the counter associated to the filter c belongs to. A filter matches an event when all its constraints are satisfied, i.e., when the associated counter is equal to the number of its constraints. If a filter matches an event so does the predicate it belongs to. Accordingly, the event can be forwarded to the interface exposing it.

Data Structures

Figure 2 shows the data structures we create and use during processing. Almost all of them are permanently stored into the GPU memory, to minimize the need for CPU-to- GPU communication during event processing. In Figure 2a we see the data structures containing information about constraints. The GPU stores table Constraints, which groups existing constraints into multiple rows, one for each name. Each element of such table stores information about a single constraint: in particular its operator (Op), its type (Type), its value (Val), and an identifier of the filter it belongs to (FilterId). Moreover, since different rows may include a different number of constraints, the GPU also stores a vector SizeC with the actual size of each row. Finally, table Constraints is coupled with map Names (<name: rowId>), stored by the CPU, which associates each attribute name with the corresponding row in Constraints.

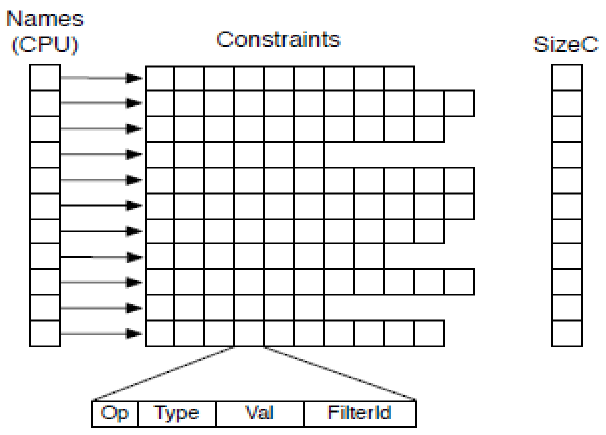


Fig 2.1 Constraints

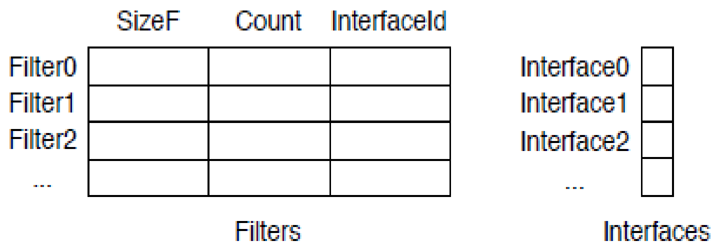


Fig 2.2 Filters And Interfaces

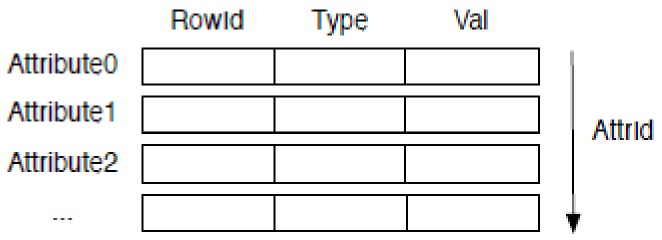


Fig 2.3 Input Data

Chapter 3
System Requirements

3.1 Hardware

To evaluate throughput of the matching algorithm, tests were taken on a machine having two NVIDIA Quadro K6000 GPGPU and two GeForceGT610. For creating the bewoulf cluster we require two system with core i7 processors and a random access memory of 16 Gigabytes.

Fig 3.1 NVIDIA GeForceGT610

Fig 3.2 NVIDIA Quadro K6000

3.2 Software

<u>Tools</u>:

- *<u>CUDA Programming Toolkit</u>*: The NVIDIA CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications. The CUDA Toolkit includes a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of your applications. You'll also find programming guides, user manuals, API reference, and other documentation to help you get started quickly accelerating your application with GPUs.
- *<u>CodeBlocks</u>*: CodeBlocks is a *free C, C++ and Fortran IDE* built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable.

<u>Technology</u>:

- *<u>C++</u>*: C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. **C++** runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
- *<u>Nvidia CUDA Compiler (NVCC)</u>*: NVCC is proprietary compiler by Nvidia intended for use with CUDA. CUDA codes runs on both the CPU and GPU. NVCC separates these two parts and sends host code (the part of code which will be run on the CPU) to a C compiler like GCC or Intel C++ Compiler (ICC) or Microsoft Visual C Compiler, and sends the device code (the part which will run on the GPU) to the GPU. The device code is further compiled by NVCC.
- *<u>MPI</u>*: Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

Chapter 4
System Design

4.1 System Architecture

- ### GPU Programming With CUDA

  CUDA is a general purpose parallel computing architecture introduced by Nvidia in November 2006. It offers a new parallel programming model and instruction set for general purpose programming on GPUs. Since parallel programming is a complex task, industry is currently devoting much effort in trying to simplify it. The most promising result in this area is OpenCL, a library that supports heterogeneous platforms, including several multicore CPUs and GPUs. However, the idea of abstracting very different architectures under the same API has a negative impact on performance: different architectures are better at different tasks and a common API cannot hide that. For this reason, although OpenCL supports the CUDA architecture, we decided to implement our algorithms in CUDA C, a dialect of C explicitly devoted to program GPUs that implement the CUDA architecture. In this section we introduce the main concepts of the CUDA programming model and CUDA C, and we brief present some aspects of the hardware implementation that play a primary role when it comes to optimize algorithms for high performance on the CUDA architecture.

4.2 Developer Perspective

The CUDA programming model is intended to help developers in writing software that leverages the increasing number of processor cores offered by modern GPUs. At its foundation are three key abstractions:
Hierarchical organization of thread groups. The programmer is guided in partitioning a problem into coarse sub-problems to be solved independently in parallel by blocks of threads, while each sub-problem must be decomposed into finer pieces to be solved cooperatively in parallel by all threads within a block. This decomposition allows the algorithm to easily scale with the number of available processor cores, since each block of threads can be scheduled on any of them, in any order, concurrently or sequentially.

Shared memories: CUDA threads may access data from multiple memory spaces during their execution: each thread has a private local memory for automatic variables; each block has a shared memory visible to all threads in the same block; finally, all threads have access to the same global memory.

Barrier synchronization: Thread blocks are required to execute independently, while threads within a block can cooperate by sharing data through shared memory and by synchronizing their execution to coordinate memory access. In particular, developers may specify synchronization points that act as barriers at which all threads in the block must wait before any is allowed to proceed. The CUDA programming model assumes that CUDA threads execute on a physically separate device (the GPU), which operates as a coprocessor to a host (the CPU) running a C/C++ program. To start a new computation on a CUDA device, the programmer has to define and call a function, called kernel, which defines a single ow of execution. When a kernel k is called, the programmer specifies the number of threads per block and the number of blocks that must execute it. Inside the kernel it is possible to access two special variables provided by the CUDA runtime: the *threadId* and the *blockId*, which together allow to uniquely identify each thread among those executing the kernel. Conditional statements involving these variables are the only way for a programmer to differentiate the execution flows of different threads. The CUDA programming model assumes that both the host and the device maintain their own separate memory spaces. Therefore, before invoking a kernel, it is necessary to explicitly allocate memory on the device and to copy there the information needed to execute the kernel. Similarly, when a kernel execution completes, it is necessary to copy results back to the host memory and to deallocate the device memory.

Hardware Implementation

The CUDA architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel k, the blocks executing k are enumerated and distributed to the available SMs. All threads belonging to the same block execute on the same SM, thus exploiting fast SRAM to implement the shared memory. Multiple blocks may execute concurrently on the same SM as well. As blocks terminate new blocks are launched on freed SMs. Each SM creates, manages, schedules, and executes threads in groups of parallel threads called warps. Individual threads composing a warp start together but they have their own instruction pointer and local state and are therefore free to branch and execute independently. When a SM is given one or more blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. All threads in a warp execute one common instruction at a time. This introduces an issue that must be carefully taken into account when designing a kernel: full efficiency is realized only when all the threads in a warp agree on their execution path. If threads in the same warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Inside a single SM, instructions are pipelined but, differently from modern CPU cores, they are executed in order, without branch prediction or speculative execution. To maximize the utilization of its computational units, each SM is able to maintain the execution context of several warps on chip, so that switching from one execution context to another has no cost. At each instruction issue time, a warp scheduler selects a warp that has threads ready to execute (not waiting on a synchronization barrier or for data from the global memory) and issues the next instruction to them. To give a more precise idea of the capabilities of a modern GPU supporting CUDA, we provide some details of the Nvidia GTX 460 card we used for our tests. The GTX 460 includes 7 SMs, which can handle up to 48 warps of 32 threads each (for a maximum of 1536 threads). Each block may access a maximum amount of 48KB of shared memory implemented directly on-chip within each SM. Furthermore, the GTX 460 offers 1GB of GDDR5 memory as global memory. This information must be carefully taken into account when programming a kernel: shared memory must be used as much as possible, since it may hide the higher latency introduced by access to global memory. However, shared memory has a limited size, which significantly impacts the design of algorithms by constraining the amount of data that can be shared by the threads in each block.

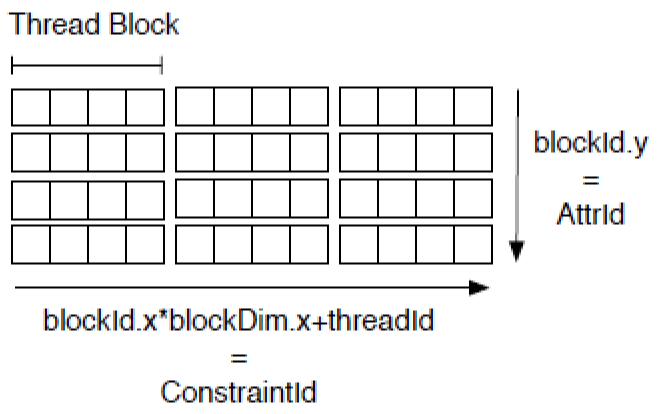


Fig 4.1 Organisation of Blocks And Threads

<u>MPI</u>

MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

MPI is not sanctioned by any major standards body; nevertheless, it has become a *de facto* standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory supercomputers such as computer clusters often run such programs. The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memory concept. Nonetheless, MPI programs are regularly run on shared memory computers. Designing programs around the MPI model (contrary to explicit shared memory models) has advantages over NUMA architectures since MPI encourages memory locality.

Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers, with sockets and Transmission Control Protocol (TCP) used in the transport layer.

Beowulf Cluster

A **Beowulf cluster** is a computer cluster of what are normally identical, commodity-grade computers networked into a small local area network with libraries and programs installed which allow processing to be shared among them. The result is a high-performance parallel computing cluster from inexpensive personal computer hardware.

The name *Beowulf* originally referred to a specific computer built in 1994 by Thomas Sterling and Donald Becker at NASA. The name "Beowulf" comes from the main character in the Old English epic poem *Beowulf*, which Sterling chose because the poem describes its eponymous hero as having "thirty men's heft of grasp in the gripe of his hand".

No particular piece of software defines a cluster as a Beowulf. Beowulf clusters normally run a Unix-like operating system, such as BSD, Linux, or Solaris, normally built from free and open source software. Commonly used parallel processing libraries include Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). Both of these permit the programmer to divide a task among a group of networked computers, and collect the results of processing. Examples of MPI software include OpenMPI or MPICH. There are additional MPI implementations available.

Beowulf is a multi-computer architecture which can be used for parallel computations. It is a system which usually consists of one server node, and one or more client nodes connected via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running a Unix-like operating system, with standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software like the FreeBSD, Linux or Solaris operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). The server node controls the whole cluster and serves files to the client nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a Beowulf system are dumb, the dumber the better. Nodes are configured and controlled by the server node, and do only what they are told to do. In a diskless client configuration, a client node doesn't even know its IP address or name until the server tells it.

One of the main differences between Beowulf and a Cluster of Workstations (COW) is that Beowulf behaves more like a single machine rather than many workstations. In most cases client nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged into the cluster, just like a CPU or memory module can be plugged into a motherboard.

Beowulf is not a special software package, new network topology, or the latest kernel hack. Beowulf is a technology of clustering computers to form a parallel, virtual supercomputer. Although there are many software packages such as kernel modifications, PVM and MPI libraries, and configuration tools which make the Beowulf architecture faster, easier to configure, and much more usable, one can build a Beowulf class machine using a standard Linux distribution without any additional software. If you have two networked computers which share at least the /home file system via NFS, and trust each other to execute remote shells (rsh), then it could be argued that you have a simple, two node Beowulf machine.

4.4 UML Diagrams

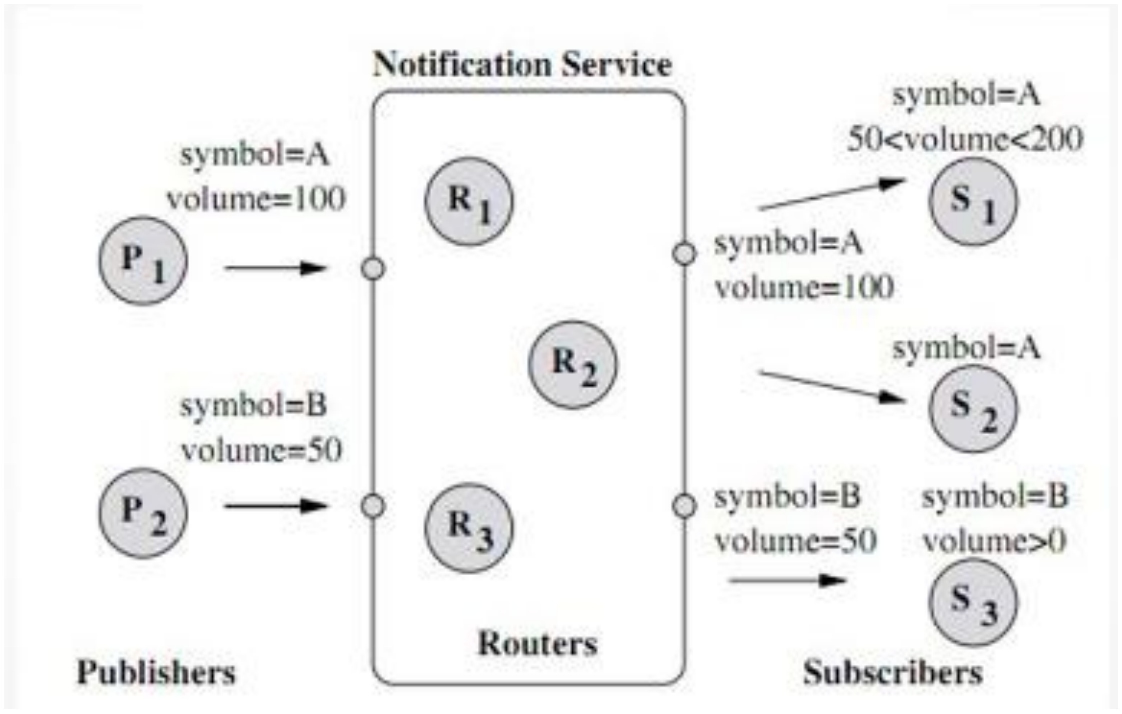


Fig 4.2 Use Case Diagram (Pub-Sub System)

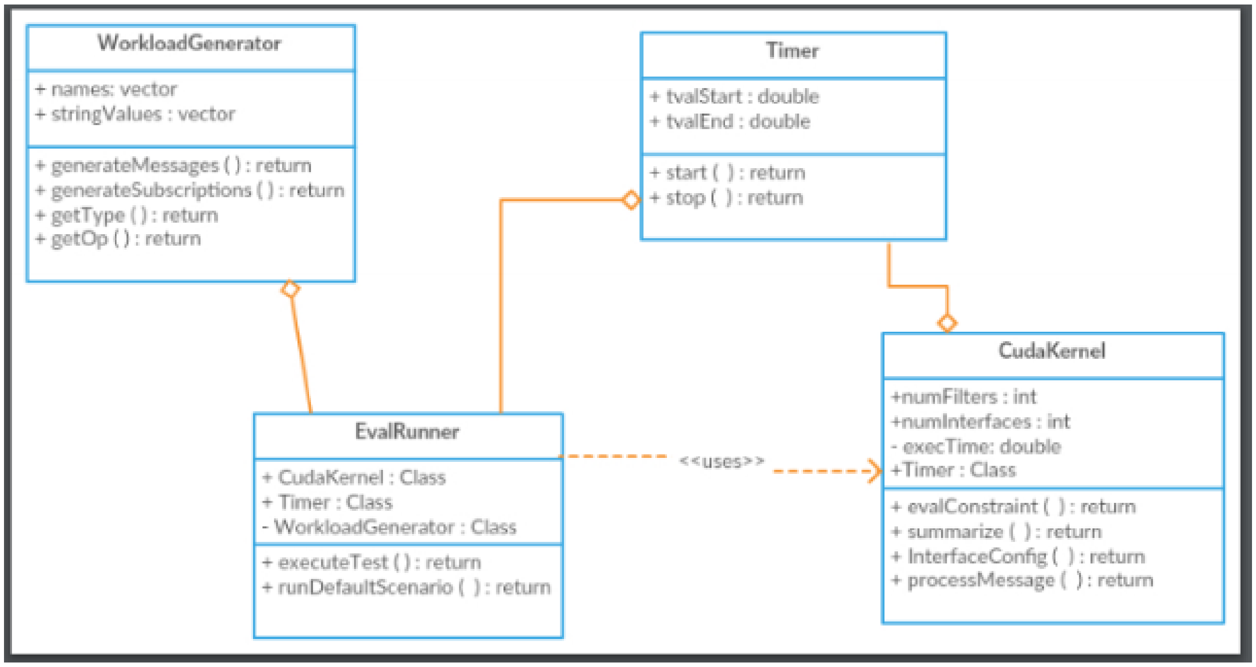


Fig 4.3 Class Diagram

Chapter 5

Methodology

In this section, we present the two parallelization techniques using multiple GPGPU for event processing: multiple events independent processing (MCCM for parallel events), single event collaborative processing (MCCM for single event). The aim of MCCM (parallel events) is to increase throughput while MCCM (single event) is used to reduce matching time of a single event.

- *Multiple Events Independent Processing (MCCM Parallel Events)*

  The aim of this parallelization technique is to increase system throughput (i.e., number of events processed per second). GPGPUs work independently on separate events to reduce the total matching time and thus increase throughput. A distinct event is given to every GPGPU for processing. For each event, GPGPU executes matching algorithm. Once the event is processed, it fetches the next available events from input stream. Minimal synchronization is needed in this approach. Following terminologies are used in the steps below: device_input, device_interface refer to respective input and interface array on device. Input[i] and interface[i] refer to input and interface array on host for i^{th} GPGPU.

  In this experimentation, single CPU thread manages the multiple GPGPU. cudaSetDevice function sets the current GPGPU and cuda calls are issued to current device. Here data pertaining to Constraints and Interfaces tables on host are transferred to the memory of every GPGPU asynchronously using default stream of GPGPU. As every GPGPU processes event individually, Input table is created for each event and transferred asynchronously to respective GPGPU for processing. The *evalConstraint* kernel is launched on every GPGPU, to process the events simultaneously. After processing, results are copied to interface array maintained on CPU for every GPGPU asynchronously. Finally we call cudaDeviceSynchronize operation for each device to confirm about completion of device operations before calculating results.

- *Single Event Collaborative Processing (MCCM Single Event)*

  The aim of this parallelization technique is to improve matching time for a single event. We use following terminologies. Offset is calculated by dividing total number of interfaces by number of GPGPUs. device_input, device_interface refer to respective input and interface array on device. host_interface refers to interface array on host.

  In this approach we divide the interfaces among number of GPGPU. Filter and Constraints tables are created for each GPGPU. The Filter and Constraints table will consists of filters corresponding to Interfaces assigned to particular GPGPU. These Filter and Constraints table are then respectively copied to GPGPU asynchronously. For an incoming event, Input table is created on CPU and then transferred to every available GPGPU. EvalConstraint kernel is called for every GPGPU simultaneously. We used offset to collect results from multiple GPUs asynchronously in single interface array. This is how we merge results from different GPGPUs. Finally we perform cudaDeviceSynchronizaiton operation to check whether all operations in default cuda streams are over for each GPGPU. Uneven distribution of filters in interfaces leads to minor degradation in performance due to difference in event matching time of individual device.

- *MPI GPU Approach*

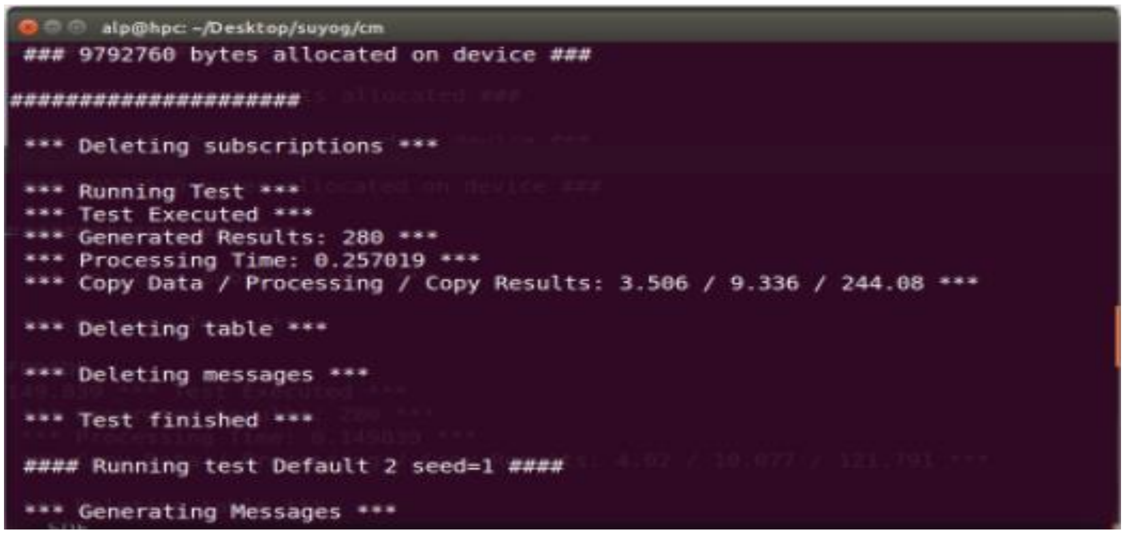
The aim of this technique is to increase system throughput (i.e., number of events processed per second). In this approach we have multiple client nodes and a single master node which manages the Event distribution work. On the master node there is a master process and other processes which handle a single GPGPU assigned to them. On client node we create processes equal to number of GPGPUs available on that system. Now the master process sends Events from input to other processed. The other processes receive the event and then run the kernel for matching. After this operation it sends results to master process. The master process is always in listening mode and as soon as it receives the result from any process it send an event to the same process from which the results were received asynchronously.

This approach fully utilizes all available resources. The main overhead in this approach is the communication time required for send and receiving over the network.

Chapter 6
Implementation

CCM

CCM algorithm is based on counting algorithm, prominently used for matching in publish subscribe systems. Data model considered for our experimentation is common for event based systems. CCM algorithm evaluates multiple constraints in subscriptions for a given attribute in an event simultaneously, by using GPGPU cores. In CCM, CPU computes the NameVector (NVe), which maps each distinct constraint name to a single bit of small bit vector. CPU creates Filters and constraints tables, to organize the constraints of the filters into five data structures namely ConstrOp, ConstrVal, ConstrFilterId, ConstrBF, and NumConstr, according to constraint name. CPU builds data structure named Filters and Interface to maintain information about filter-size, filter-count and interface-Id to which the filter belongs. Interface is an array, which is the result of CCM algorithm, depicts matched interfaces by setting corresponding index of the array to one. For every event to be processed, CPU builds table Input. It includes one line for each attribute in event e. For each attribute a in e, each line of input table stores the value of a, its type, the number of constraints having the same name as a, and the pointers (in the GPGPU memory) to the rows of Filters and constraint table that are relevant for a. All these data structures are transferred to the GPGPU for processing an event. CCM launches a kernel named *evalConstraint*, which uses thousands of GPGPU threads to evaluate constraints in parallel. Each thread evaluates a single attribute a, of e against a single constraint c. After that, the results of the computation (i.e., the Interfaces array) have to be copied back to the CPU memory and the FiltersCount and Interfaces structures have to be reset for the next event.



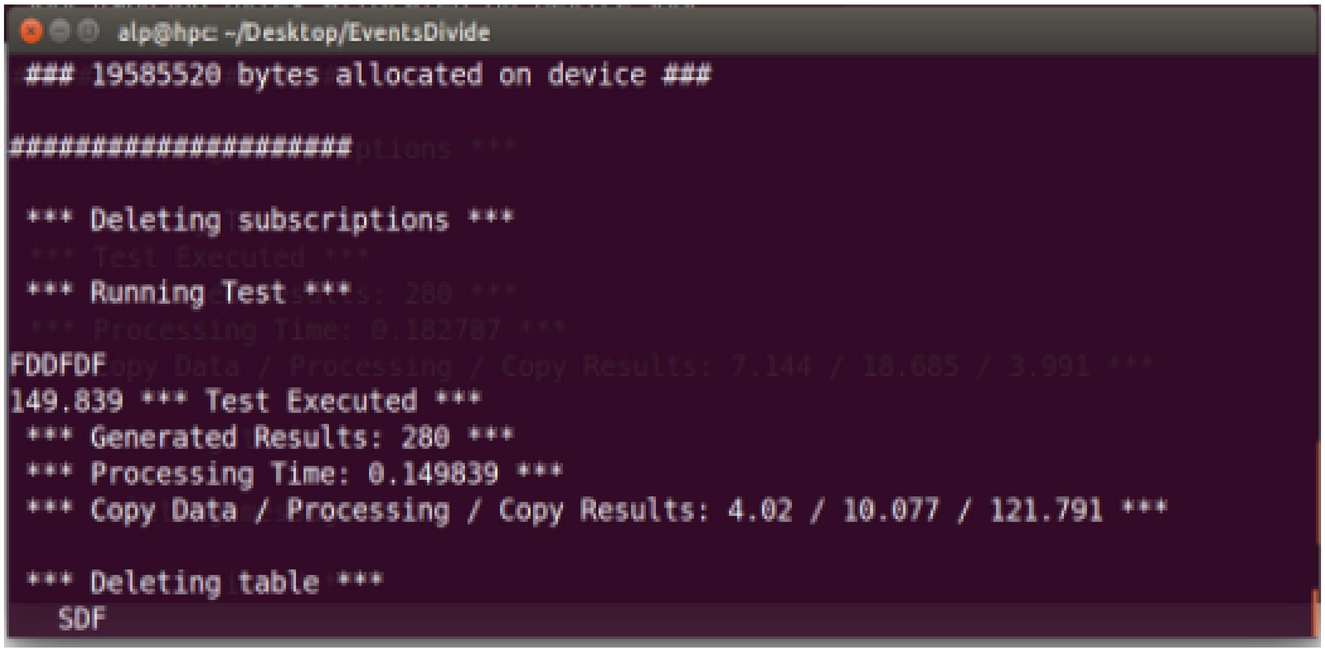
```
alp@hpc: ~/Desktop/suyog/cm
### 9792760 bytes allocated on device ###

#####################

 *** Deleting subscriptions ***

 *** Running Test ***
 *** Test Executed ***
 *** Generated Results: 280 ***
 *** Processing Time: 0.257019 ***
 *** Copy Data / Processing / Copy Results: 3.506 / 9.336 / 244.08 ***

 *** Deleting table ***

 *** Deleting messages ***

 *** Test finished ***

 #### Running test Default 2 seed=1 ####

 *** Generating Messages ***
```

Fig 7.1 Implementation results for CCM

<u>MCCM</u> (Parallel Events)

In case of multiple events independent processing MCCM (Parallel Events), following are the steps to carry out the matching algorithm on multiple GPGPUs.

```
1. Filters and Constraints tables on host are copied to memory of every GPGPU asynchronously.
2. Incoming events are stored in Input stream.
3. For each GPGPU 'i'
       cudaSetDevice(i)
       create table input[i] for event to be processed.
        cudaMemcpyAsync(device_input, input[i], cudaMemcpyHostToDevice)
       evalConstraint<<<NUM_BLOCKS,  NUM_THREADS>>>
    End for
4. For each GPGPU ‘i’
        cudaSetDevice(i)
        cudaMemcpyAsync( interface[i], device_interface, cudaMemcpyDeviceToHost)
       cudaDeviceSynchronize();
       cudaMemset(device_interface);
   End for
```



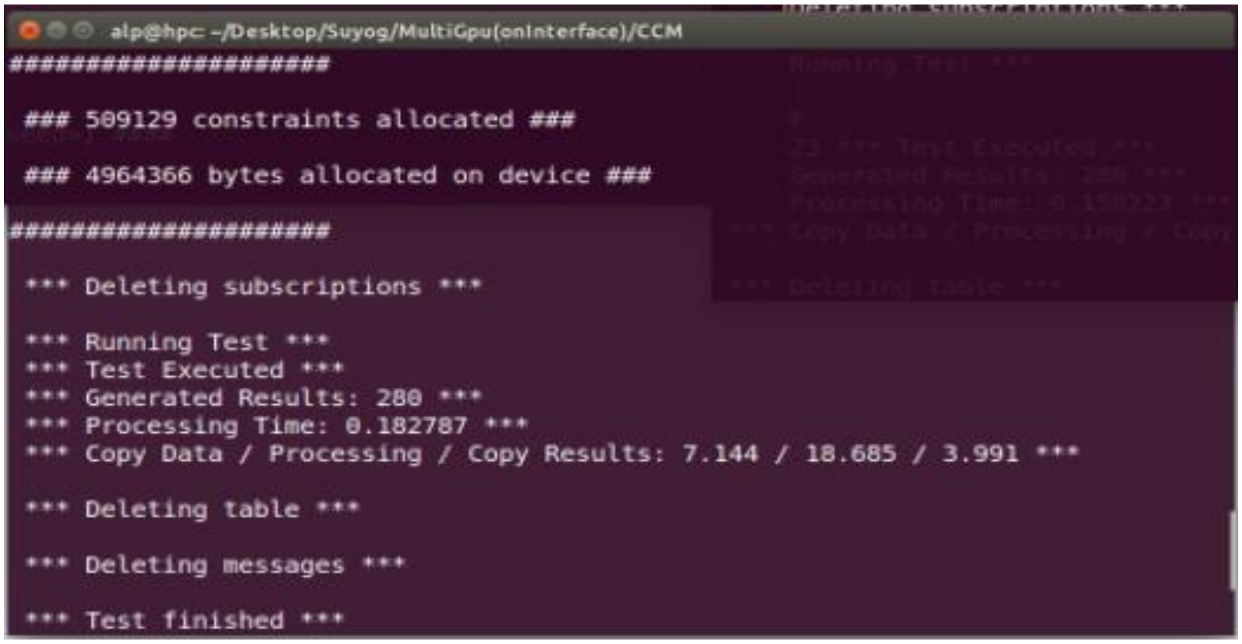
```
alp@hpc: ~/Desktop/EventsDivide
 ### 19585520 bytes allocated on device ###

#####################

 *** Deleting subscriptions ***

 *** Running Test ***

FDDFDF
149.839 *** Test Executed ***
 *** Generated Results: 280 ***
 *** Processing Time: 0.149839 ***
 *** Copy Data / Processing / Copy Results: 4.02 / 10.077 / 121.791 ***

 *** Deleting table ***
   SDF
```

Fig 7.2 Implementation results for MCCM (Parallel Events)

<u>MCCM</u> *(Single Event)*

In case of single event collaborative processing (MCCM Single Event), following are the steps to carry out the matching algorithm on multiple GPGPUs.

```
1. Interfaces on host are divided equally among number of GPGPUs available.
2. Filters and Constraint tables corresponding to interface are created for individual GPGPUs.
And transferred to GPGPU memory.
3. Incoming events are stored in Input stream.
4. Input table (host_input) is created for an event to be processed.
5. For each GPGPU 'i'
       cudaSetDevice(i)
        cudaMemcpyAsync(device_input, host_input, cudaMemcpyHostToDevice)
       evalConstraint<<<NUM_BLOCKS,  NUM_THREADS>>>
    End for
4. For each GPGPU ‘i’
        cudaSetDevice(i)
        cudaMemcpyAsync( host_interface+i*offset, device_interface,
cudaMemcpyDeviceToHost)
       cudaDeviceSynchronize();
       cudaMemset(device_interface);
  End for
```



```
alp@hpc: ~/Desktop/Suyog/MultiGpu(onInterface)/CCM
#####################

 ### 509129 constraints allocated ###

 ### 4964366 bytes allocated on device ###

#####################

 *** Deleting subscriptions ***

 *** Running Test ***
 *** Test Executed ***
 *** Generated Results: 280 ***
 *** Processing Time: 0.182787 ***
 *** Copy Data / Processing / Copy Results: 7.144 / 18.685 / 3.991 ***

 *** Deleting table ***

 *** Deleting messages ***

 *** Test finished ***
```

Fig 7.3 Implementation results for MCCM (Single Event)

To evaluate the latency of pure matching, we have taken same default scenario of whose parameters are listed in Table 1, and used it as a starting point to build a number of different experiments, by changing the various parameters one by one and measuring how this impacts the performance of CCM, MCCM(Parallel events), MCCM(single event).

Default scenario: Table 7.1 shows the processing times measured by the algorithms under analysis in the default scenario.
Processing Time in the Default Scenario**:**

CCM	**MCCM(Parallel Events)**	**MCCM(Single Event)**	**MPI-GPGPU**
0.25245ms	**0.1505ms**	**0.1798ms**	**0.1254ms**

Table 7.1: Processing Time in Default Scenario

Under this load, If we consider all algorithms, CCM requires 0.25245 ms and MCCM (parallel events) 0.1505 ms, MCCM (single event) 0.1798 ms providing respectively a improvement of 0.10195 ms (40%) and 0.07265 ms (33%) with respect to CMM. To test the real benefits of MultiGPGPU algorithm we made changes in parameters of default scenario.

Numbers of Interfaces: Fig. 7.4 shows how performance changes with the increasing number of interfaces. It is observed that beyond certain limit of interfaces CCM, doesn't provide results. The reason may be memory limit of single GPU. MCCM (Single Event) produces result for large values of interfaces. Processing time is low for algorithm MCCM (Single event) as it divides the interfaces among available GPGPUs.

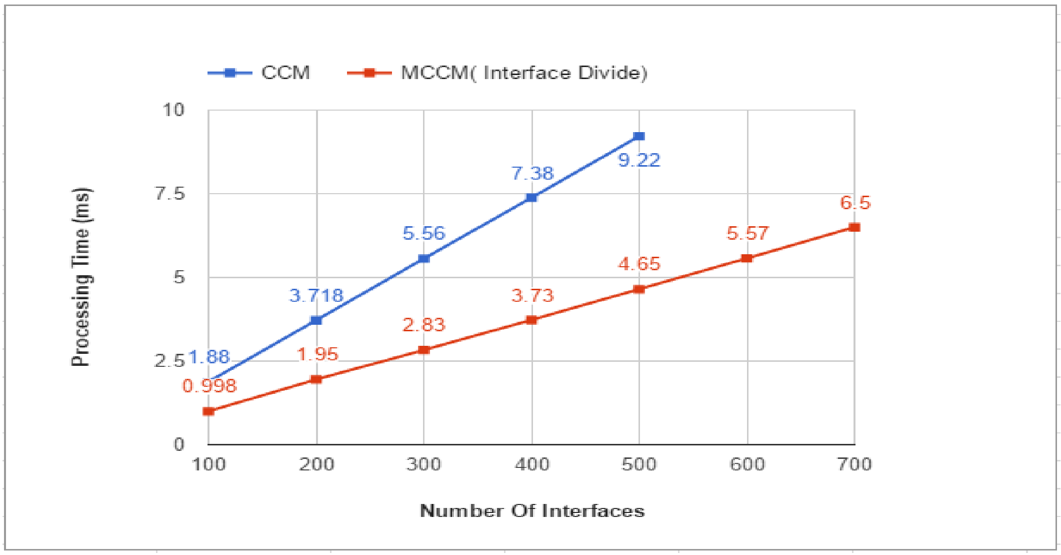


Figure 7.4 Number of Interfaces

Number of Filters per Interface: Fig. 7.5 shows how performance changes with the number of filters per interface. Increasing such number also increases the overall number of constraints, and thus the complexity of matching. Accordingly, all the algorithms show growing, processing times. This scenario emphasizes the advantages of parallel processing when interfaces are divided among multiple GPGPU. MCCM (single event) exhibits good for performance with increasing number of filters per interfaces.

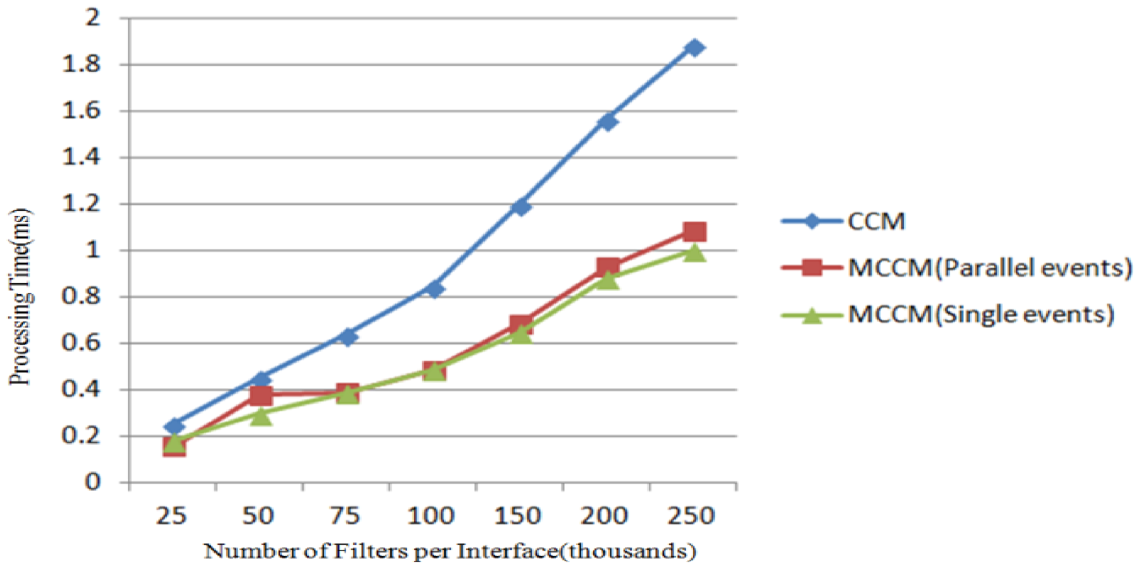


Figure 7.5 Number of filters per interface.

Random Number of Filters per Interface: To test our multiGPGPU algorithm we made random distribution of number of filters in interfaces. Min and Max values set for number of filters per interface are varied from 5 to 2,27,500. It is observed that MCCM (single event) still performs better than CCM by giving improvement of 40%.

MPI GPU Approach

This approach takes benefits of the CCM algorithm and combines it with high level parallelism using MPI. A **Beowulf cluster** is a computer **cluster** of what are normally identical, commodity-grade computers networked into a small local area network with libraries and programs installed which allow processing to be shared among them.

MPI GPU Algorithm

```
1. Get number of GPU's available on each system connected in cluster.
2. Assign each device to process on system.
3. If master process
        Get all subscriptions.
       Transfer them to all process
   Else
       Receive subscriptions from master process.
4. Transfer subscriptions to GPU's assigned to the process.
5. If master process
      while all events not processed
          Receive matched interfaces from 'x' process
          Send incoming event to 'x' process
6. Else
      while(true)
          Receive event from master process
          If number of attributes in event equals zero then break;
              Process Event
              Send matched interfaces to master process.
```

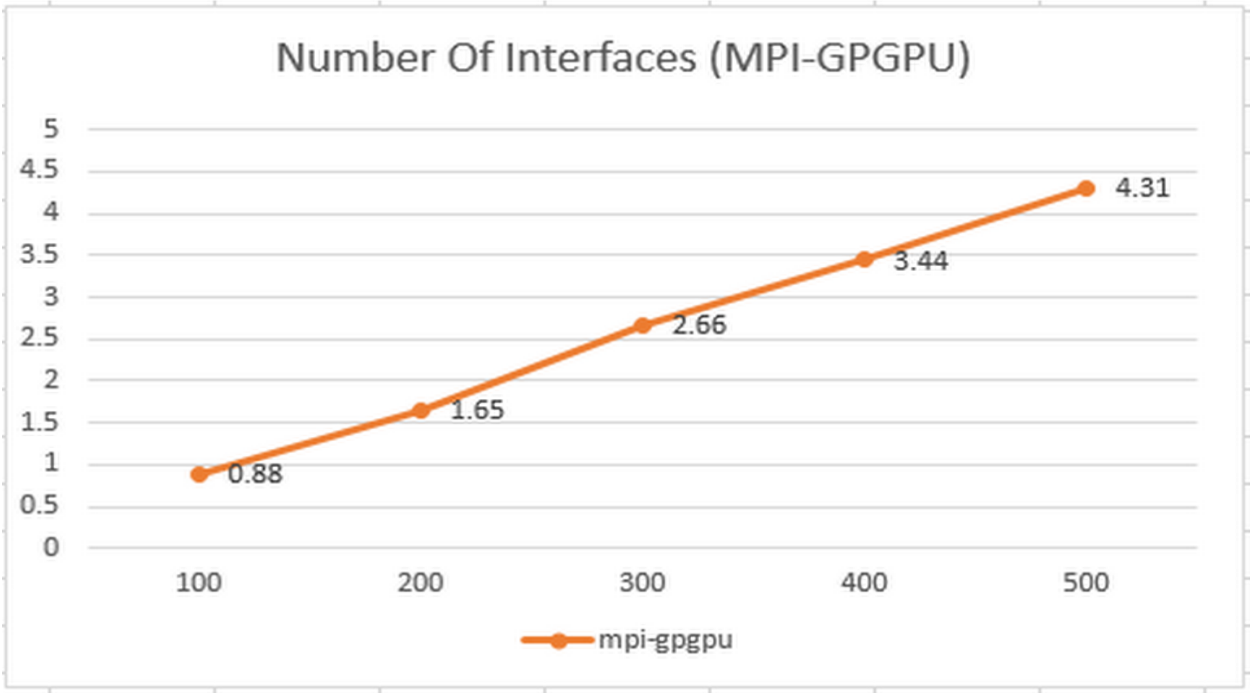
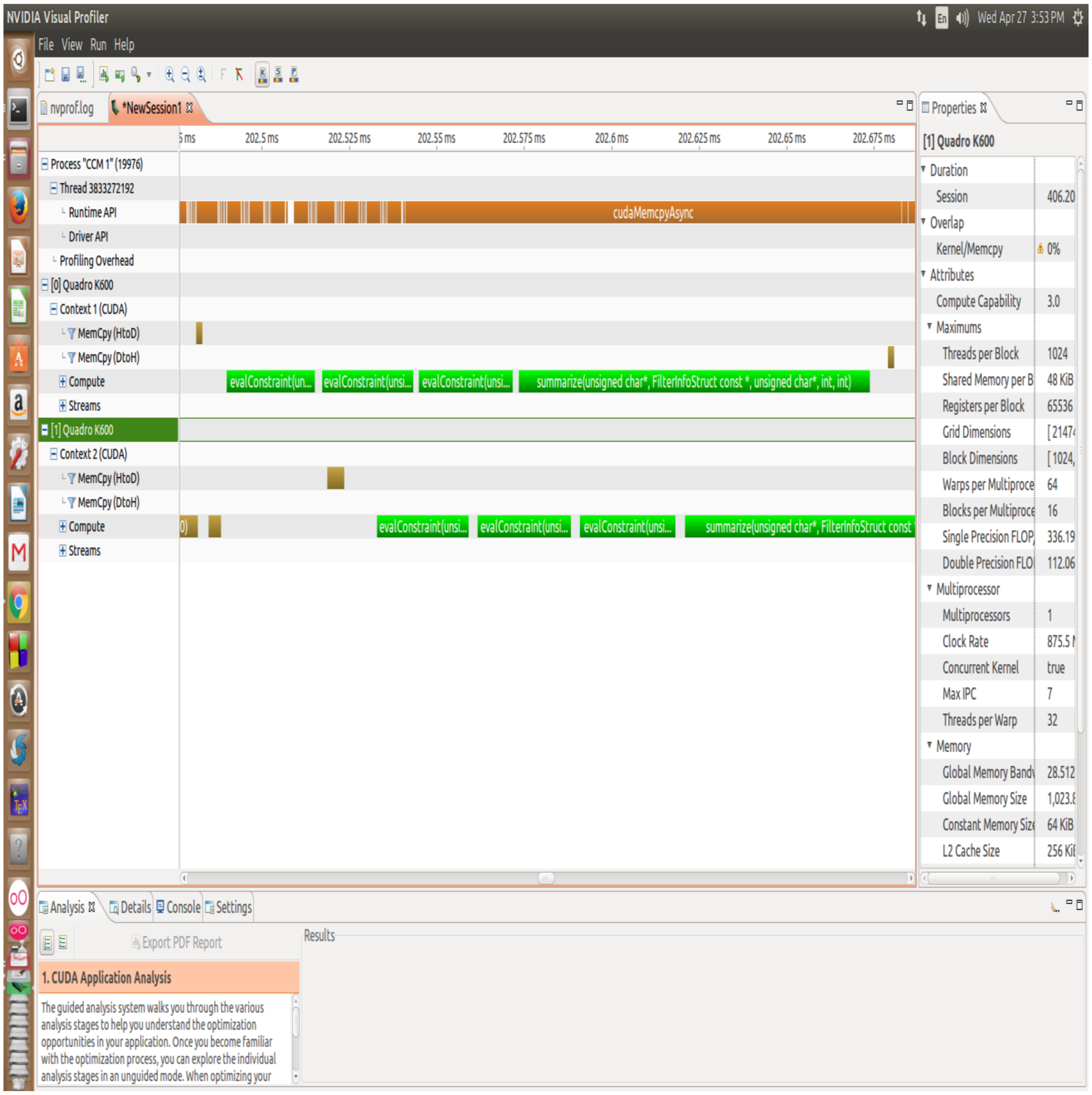


Fig. 7.6 Number of Interfaces (MPI-GPGPU)



[1] Quadro K600

Duration	
Session	406.20
Overlap	
Kernel/Memcpy	0%
Attributes	
Compute Capability	3.0
Maximums	
Threads per Block	1024
Shared Memory per B	48 KiB
Registers per Block	65536
Grid Dimensions	[2147
Block Dimensions	[1024,
Warps per Multiproce	64
Blocks per Multiproce	16
Single Precision FLOP	336.19
Double Precision FLO	112.06
Multiprocessor	
Multiprocessors	1
Clock Rate	875.5 M
Concurrent Kernel	true
Max IPC	7
Threads per Warp	32
Memory	
Global Memory Bandw	28.512
Global Memory Size	1,023.8
Constant Memory Size	64 KiB
L2 Cache Size	256 KiB

1. CUDA Application Analysis

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your

Fig 7.7 NVIDIA Visual Profiler

Chapter 7

Conclusion

Here we conclude that, multiple-GPGPUs help in improving performance in terms of throughput and matching time. Though added resource helps improving performance, decomposition of data, synchronization of GPGPU plays major role in performance improvement. We have also tested our algorithm on MPI+GPGPU system.

References

[1] AMD. ATI Stream. http://www.amd.com.

[2] NVIDIA. CUDA. http://www.nvidia.com.

[3] Alessandro Margara, Gianpaolo Cugola. 2014. High-Performance Publish-Subscribe Matching Using Parallel Hardware.In IEEE Transactions on Parallel and Distributed Systems Volume:25, Issue: 1 January 2014

[4] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. 2009. Parallel Event Processing for Content-Based Publish/Subscribe Systems. In (DEBS '09), pp. 8:1-8:4, 2009.

[5] Medha A. Shah, D.B.Kulkarni.2015. Storm Pub-Sub: High Performance, Scalable Content Based Event Matching System Using Storm. In IPDPSW'2015

[6] Alessandro Margara , Gianpaolo Cugola. High-Performance Publish-Subscribe Matching Using Parallel Hardware, IEEE Transactions on Parallel and Distributed Systems Volume:25 Issue:1 January 2014

[7] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen, Parallel Event Processing for Content-Based Publish/Subscribe Systems (DEBS '09), pp. 8:1-8:4, 2009.

[8] Raphaël Barazzutti, Pascal Felber. StreamHub: A Massively Parallel Architecture for High-Performance Content-Based Publish/Subscribe DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

[9] Medha A. Shah, D.B.Kulkarni. Storm Pub-Sub: High Performance, Scalable Content Based Event Matching System Using Storm. In IPDPSW'2015

[10] Zhaoran Wang, Xiaotao Chang. Pub/Sub on Stream: A Multi-Core Based Message Broker with QoS Support DEBS 2012, July 16–20,2012, Berlin, Germany July 2012.Forman, G. 2003. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.* 3 (Mar. 2003), 1289-1305.

[11] Enabling QoS Support for Multi-Core Message Broker in Publish/Subscribe System , Medha Shah, D.B.Kulkarni Advance Computing Conference (IACC), 2014 IEEE International conference.

[12] K.H. Tsoi, I. Papagiannis, M. Migliavacca, W. Luk, and P Pietzuch. Accelerating Publish/Subscribe Matching on Reconfigurable Supercomputing Platforms.Proc. Many-Core and Reconfigurable Supercomputing Conf., 2010.

[13] Han Xiao,Yu-Pu Song and Qing-Lei Zhou. Multi GPU Accelerated Parallel Algorithm of Wallis Transformation for Image Enhancement. International Journal of Grid and Distributed Computing Vol.7, No.2 (2014), pp.99-114

[14] G. Muhl, L. Fiege, and P. Pietzuch. Distributed Event-Based Systems. Springer, 2006.

[15] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical report, DIS, La Sapienza, 2005.

[16] G. Muhl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In MASCOTS, 2002.

[17] G. Cugola and G. Picco. REDS: A Reconfigurable Dispatching System. In SEM, pages 9—16, Portland, 2006. ACM Press

[18] T. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. ACM TODS'94.

[19] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for fast pub/sub systems. SIGMOD'01

[20] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina. Indexing Boolean expressions. In VLDB'09.

[21] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In PODC'99