GPU EXECUTION TRACING AND COMPRESSION

by

Sahar Azimi Moghaddam, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2017

Committee Members:

Martin Burtscher, Chair

Apan Qasem

Ziliang Zong

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

To my father, a strong and gentle soul who taught me to believe in hard work and has been my inspiration throughout my life. To my mother, who taught me to trust in Allah, myself and my dreams. I could not have done this without your faith, support and constant encouragement. To my mentor, this could not have been possible without you. Thank you for all of your support. To my beloved husband, who has been so supportive along the way. I cannot thank you enough.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Martin Burtscher, without whose help and support this work would never have been possible. I am grateful for his patience and guidance on each and every step of the journey. I learned a great deal during our individual and group sessions. I would like to thank Dr. Qasem and Dr. Zong for agreeing to serve on my committee. I am particularly grateful to those who teach and share their knowledge broadly.

I would like to thank Texas State University, and in particular the ECL lab established by Dr. Burtscher, which I am proud to be a member of.

This project is supported by the National Science Foundation under award #1438963 and by hardware donations from NVIDA Corporation.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
GPU	Graphics Processing Unit
MIS	Maximal Independent Set
DMR	Delaunay Mesh Refinement
MST	Minimum Spanning Tree
fr	Fractal
nb	N-Body

ABSTRACT

Program tracing is widely used for debugging and performance optimization. Whenever a program is traced, the overhead in terms of extra runtime and in terms of storage for the generated trace information are a concern. These concerns are greatly exacerbated on GPUs due to the large amount of parallelism. In fact, GPUs provide such massive parallelism that conventional tracing approaches either fail or only manage to trace very few events per thread. Hence, we need not only a low-overhead but also a space-efficient approach to make detailed tracing possible on GPUs. To the best of my knowledge, none of the existing GPU tracing tools support both. Thus, in this thesis, I developed an execution tracing tool for GPUs called ECL-Tracer that is light-weight and immediately compresses the generated trace data before they are stored.

1. INTRODUCTION

Due to their high computational power, low cost, and energy efficiency, GPUs are increasingly used in various disciplines such as simulations, solving complex problems, climate modeling, drug discoveries, data analysis, etc. Unfortunately, writing and tuning programs for GPUs is more difficult than for CPUs. Tracing tools have the potential to play an important role in the development of high-performance GPU codes, but no whole-program tracing tools exist so far. The main reason is the massive parallelism of GPUs, which can simultaneously hold up to almost 50,000 threads. Tracing just a single event per thread thus produces up to 50,000 events that need to be recorded. As a consequence, I/O bandwidth quickly becomes a bottleneck when writing such immense amounts of trace data to disk or even just to memory. To make the tracing of a large number of events per thread and per second possible, the trace data must be compressed right away and quickly before it is stored.

1.1 Large Size Traces

Although tracing events helps us detect problems, growing trace-file sizes complicate the analysis and management of data in large-scale systems. In this section, some factors that contribute to large traces are explained.

- **Number of threads:** This factor represents the number of threads captured for tracing. Collecting trace data from a larger number of threads results in correspondingly larger trace files.

- **Number of event parameters:** This factor represents the parameters recorded as part of an event. The most typical parameters include a thread identifier, a time stamp, memory transfer information, device specific information, and a type identifier [1]. In my thesis, I pass the function call location as the event parameter.
- **Granularity:** The granularity of detail recorded depends on the granularity at which the events are traced. In my project, events are recorded wherever the user places a call to the ECL-Tracer in the source code. I placed such a call at every control-flow decision point in the code.
- **Problem size:** This factor is the input size given to the program to be traced. Depending on the program, larger inputs may need more operations to be processed, which leads to generate larger traces [1].

Large traces prevent us from tracing for a long time as either the memory is filled soon or the amount of data to be transferred will become an issue. These problems will become worse in the future due to the rapid increase in use of accelerator-based devices. Equally important, within the next few years, we will witness a considerable rise in the amount of data to be processed by these applications, hence we need more memory and space to store the traces.

To overcome the problems mentioned above, we need a tracing tool and trace-compression mechanism to reduce the generated trace file size as the need for such a tool is greater than ever before.

1.2 Contributions

This thesis makes the following contributions:

1. A light-weight, portable GPU tracing tool called ECL-Tracer
2. This tool provides efficiency in time and space.
3. This tool incorporates a novel, incremental compression algorithm that compresses the traces at runtime and before writing them to memory.

1.3 Results

The ECL-Tracer works well for most parallel programs with an overhead of 1.03 to 5.27. It compresses data by up to a factor of 385 and is faster than standard compression algorithms such as Bzip2. Moreover, the ECL-Tracer is portable and works on different GPU architectures.

1.4 Outline

In this thesis, Chapter 2 gives a brief overview of the background of tracing and data compression. In Chapter 3, prior work is explained. I introduce the ECL-Tracer tool and describe the design and implementation of the tool in Chapter 4. Chapter 5 begins with the methodology and the configuration on which the experiments were conducted and evaluated. Results are discussed in Chapter 6. Finally, Chapter 7 presents the summary and future work.

2. BACKGROUND

In this chapter I explain a brief background on tracing, the common techniques for implementing it, and data compression techniques.

2.1 Tracing Definition and Applications

To put it simply, when we go through a program and check the values of different variables and record the output, we are tracing the program. In software engineering, information about a program execution is collected and recorded with the help of logging technique. This method is generally called tracing.

A trace program is usually referred to as a utility program that captures the sequence of executing events in another program [1]. A tracing routine provides a chronological record of the execution of a computer program [1]. The tracing is often performed to find out what a program exactly executes and where the origins of problems are [2]. Moreover, event tracing is a powerful method for performance analysis [3]. In particular, in parallel programs we can monitor how threads interact while communicating and study the effect of concurrent activities on the performance of each other.

2.1.1 Tracing techniques

Tracing techniques are different based on their application and the data they capture. However, one of the common tracing techniques in software programming that I used in ECL-Tracer is called instrumentation, which is explained as following.

Instrumentation

The Instrumentation technique is implemented by means of adding trace code instructions to the main application. This technique allows us to track the execution of specific sections in a code [4]. In other words, it can receive and collect informative data and write it to the specified target such as the screen output or a file stored on the disk. One popular way of instrumentation is to record the information obtained from function calls, i.e. the location of the function in the source code.

2.2 Compression

In general, the process of reducing the size of a given data file is called data compression. This is useful because it reduces the resources required to store and transmit data. The original data can be reproduced by inverting the compression (decompression). Two approaches exist for compression: lossy and lossless. In lossless data compression, no information is lost because it exploits statistical redundancy to reduce bits. Lossy method, however, eliminates some detail or less-important data to reduce the data file size [5].

As mentioned above, trace data helps us study the behavior of a program. Hence, we need the trace data to be highly accurate. Accordingly, a good tracing compression algorithm is the one that not only compresses data but also guarantees the accuracy of reproduced data after decompression. For this reason, lossy algorithms are not suitable choices as they cannot perfectly reconstruct the original data.

There are several compression techniques used in compression algorithms. One of them is called table-based compression model. In this technique, a table is generated dynamically from the earlier data from the input. Table entries are substituted for repeated strings of data. LZ77, which is one of the most popular lossless compression algorithms, uses this model. Probabilistic models are also used in compression. In these models, data is compressed based on a prediction obtained by partial matching.

2.2.1 Gzip

Gzip is a utility designed for file compression and decompression. The software was basically created to be a replacement for the compress program and compresses much better than the replaced software [6]. Gzip algorithm is based on the combination of LZ77 and Huffman coding [7] and provides lossless data compression [8].

Given an input, the LZ77 algorithm looks for the repeated strings in the input data and replaces the second occurrence of the string with a reference to the previous string. This reference is formed by a pair of values: the jump, which is the distance from the previous string, and the length of the string. Gzip emits the strings that does not repeat in the input. The Huffman algorithm is then used by Gzip to compress the matching lengths and matching distances. Huffman coding is based on the **variable-length coding method**. In this method, the more frequent characters are assigned shorter codes [9].

2.2.2 Bzip2

Bzip2 is an open-source and high-quality data compressor that uses a stack of several compression techniques. It starts with Run-length encoding (RLE) [10] and produces blocks of size between 100 and 900 KB. RLE replaces repeating data by a count and one copy of the repeated element. Then it uses Burrows–Wheeler transform [11] to convert frequently occurring character sequences into strings of identical letters. It then applies move-to-front transform [12] and Huffman coding [7]. Bzip2 is more efficient than Gzip but tends to be considerably slower [13].

3. RELATED WORK

To the best of my knowledge, all tracing tools for GPU programs do not compress their output. Moreover, there are compression algorithms for GPUs but not specifically for tracing. As a result, I discuss the prior work in two separate sections.

3.1 Tracing tools

Tracing tools for GPUs are being utilized for performance analysis [14] [15], tracing memory addresses [16], etc. As described elsewhere [17], the first GPU tracing tools, including AMD's CodeXL [18] and Nvidia's Nsight [19], were proprietary tools. Third-party tools that provide similar functionality, such as TAU [20] and VampirTrace [21], typically also support for MPI. For GPUs, these tools primarily provide a visual representation of API calls, memory transfers, and kernel execution. A recent article describes a system-wide unified CPU and GPU tracing tool (CLUST) for OpenCL applications [22]. They added an extension to the LTTng tracing tool that enables programmers to gain a better global view of OpenCL applications by using GPU tracing along with CPU tracing. MPTrace is a debugging tool for GPUs that is based on in-line tracing [23]. Another debugging tool for race condition detection [24] employs an optimal strategy to record just the minimum number of shared-memory references required to exactly replay the execution. None of these tools directly compresses the generated trace data.

3.2 GPU-Compression Tools

Several compression algorithms for GPUs have been proposed, including a parallel implementation of bzip2-like lossless compression [25]. Compression using this algorithm is slower than bzip2 but decompression is faster. Another compression algorithm for GPUs is CULZZSS-Bit [26], which exploits bit parallelism. Yet another algorithm is based on statistical and dictionary approaches to arrive at a general-purpose compression algorithm for GPUs [27]. Some compression algorithms for GPUs target floating-point data [28] [29]. However, none of these algorithms were designed specifically for compressing trace data. Regarding GPU trace compression, Goel et al. applied online stream compression to create a compact execution collector for shared-memory parallel programs and showed that their technique outperforms Gzip [30]. However, they do not compress the trace data immediately but first store it in uncompressed format. There are several trace compression algorithms for (serial) CPU execution, including a hardware-based approach for trace compression and on-the-fly trace decompression [31]. VPC3 [32] is a program trace-compression algorithm that makes use of value predictors to compress traces more efficiently. There are also approaches for both instruction and data address traces [33]. Unfortunately, none of these algorithms can be readily parallelized, making them unsuitable for GPUs.

4. DESIGN AND IMPLEMENTATION

In this thesis, I devised an efficient tracing tool for massively parallel programs running on GPUs. The tool does not incur much overhead so as not to slow down the execution to an unacceptable level. Moreover, it does not produce too much data per traced event so that a large number of events can be recorded. The basic strategy for tracing is to provide a simple interface to the programmer that allows the marking of all points in the program that should be traced, i.e., a trace event is recorded whenever a thread reaches one of these points during execution. I studied such traces from a suite of GPU programs and determined simple yet effective compression algorithms for the resulting type of data. Eventually, I implemented the best such compression algorithm directly in the tracing tool.

This chapter describes how ECL-Tracer was designed and developed to efficiently trace and compress the traced data for a massively parallel program.

4.1 GPU Parallelism

CUDA Architecture exposes GPU parallelism for general-purpose computing. On modern GPUs, CUDA provides software programmers with three granularities, warp, block, and grid.

The first level, which is called warp, is formed by grouping 32 contiguous threads together to work in lockstep. That is, threads within a warp must follow the same execution path, i.e. threads either execute the same instruction on different data in the same cycle (active threads) or they are disabled (inactive threads). In other words, threads

cannot diverge. Nonetheless, branch or thread divergence is inevitable if some threads in a warp do not fit into the criterion for an instruction. In this case, they jump to different instruction and become inactive. Inactive threads must wait while the hardware runs the active threads so that they can re-converge again. Thread divergence could result in reduction of parallelism and eventually performance loss in massively parallel applications that suffers from excessive divergent codes. Data can be exchanged between threads within a warp without an explicit synchronization.

The second level is larger than warp and is represented by thread blocks. In fact, a thread block is divided into warps. In current GPUs, block allocation limits allow up to 1024 threads per block. The programmer is responsible to choose the size of a thread block.

Grids that are third level of parallelism in current GPUs can hold up to two billion blocks. Like for thread blocks, the grid size is determined by the programmer. GPUs allow cross-block communications only through the global memory which is a shared L2 cache and because L2 cache has limited capacity (few words per thread) it is not able to provide fast communication between thousands of running threads by a GPU, therefore communication between blocks are slower than within-block communications.

One method to obtain data from lower levels of parallelism such as warps is using CUDA warp-vote functions. Generally, warp vote functions take a predicate (normally an integer) from each thread in the warp and compare those values with zero. The result of the comparisons is combined across the active threads of the warp and broadcasted a single return value to each participating thread. One of the warp vote functions that I used in this thesis is *ballot* function. This function returns an integer whose Nth bit is set only

if the result of the comparison for the Nth thread in the warp is non-zero and the Nth thread is active [34].

4.2 Data Collection

As mentioned in section 2.2, I used instrumentation technique to trace the events. This was implemented by instrumenting function calls anywhere in the code that we needed to obtain information about. In fact, by calling each of the instrumented functions we are passing parameters and record information about that specific location in the code. GPUs use threads for executing the codes, thus our functions are called by each thread reaching the function. But here are two big problems with GPUs. First of all, GPUs do not run a few threads, but around 50,000 of them run simultaneously. This will instantly create a huge amount of data that needs to be handled. Second of all, data collection can be more complicated by the fact that GPUs do *warp based execution* (cf. Section 4.1), which causes thread divergence. To handle these problems, I decided to take the *warp* as the main unit and use warp voting functions to collect data. Below I will explain how ECL-Tracer collect data from running warps.

For each running warp, ECL-Tracer records the location of the called function in the code and the number of active and inactive threads in the warp. At first, all the trace function calls must be assigned a unique ID and added to the source code wherever data needs to be collected such as locations ❶,❷, and ❸ in the pseudocode on the next page.

```
void traceLoc(unsigned char location) {
  // warp voting function
  int bmp = __ballot(location);
  // CUT: converting bmp to sequences of bytes
  for (int i = 0; i < 4; i++) {
    b = bmp & 0xFF;
    // passing bytes to next compression component
    LZ(b);
    bmp >>= 8;
  }
}
void myKernel(int nodes) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int a[nodes];

  traceLoc(1);
  if (index < nodes) {
    traceLoc(2);
    for (int i = 0; i < (nodes / 2); i++) {
      traceLoc(3);
      a[index] += i;
    }
  }
}
```

Then in the initialization step, the ECL-Tracer creates a counter for each running warp to keep track of the number of times a trace event is generated by a warp. Whenever a function is called, the ID associated with the function along with a bitmap of active threads ❹ in the warp are obtained, converted to a sequence of bytes ❺, and passed to the compression phase on-the-fly ❻. For instance, the trace function to be revoked from location ❶ will be assigned number 1. When warp zero calls *traceLoc (1)*, "1" will be

passed to the function as an argument. The trace function also records the bitmap of active threads using the value returned from __ballot() function. This value is a 32-bit integer in which each bit position represents a thread. If a warp in location ❷ calls *traceLoc(2)* and the warp vote function returns 15, it means that only warps zero to 3 are active and executing the *traceLoc(2)*. This value first will be converted to a sequence of bytes ❺ and then passed to next compression component ❻.

4.3 Compression

Since a large number of threads execute the trace functions, the recorded data tend to be very large, thus transferring and writing them to the secondary storage soon will be an I/O and storage bottlenecks. Hence, I decided to design a simple yet effective compression algorithm.

One approach is to compress the trace data after it completely generated. However, it is not an efficient way as it still needs a huge amount of storage for the complete trace. To overcome this problem, I took a better approach which is to incrementally compress the trace data as they are being generated. Implementing this approach required a compression algorithm that not only maintains a good compression ratio, but also does not incur much overhead on the system to not to slow down the execution to an unacceptable level.

To find a fulfilling algorithm I used a tool called CRUSHER [29] which is a tool that automatically generates high-performance lossless compression algorithms by

chaining components. This tool could be limited to only synthesize GPU-friendly compression algorithms.

CRUSHER was run on my previously generated traces from different parallel programs. Based on what CRUSHER reported, a chain of three components called CUT, LZ, and RLE would be a good compression algorithm. CUT is simply a type cast that converts a block of words into a block of bytes. LZ and RLE take sequence of values as input and output transformed sequences. In general, the output of one component is the input for the next component and the output of RLE will be the compressed trace. Figure1 illustrates the data flow through the compression components.

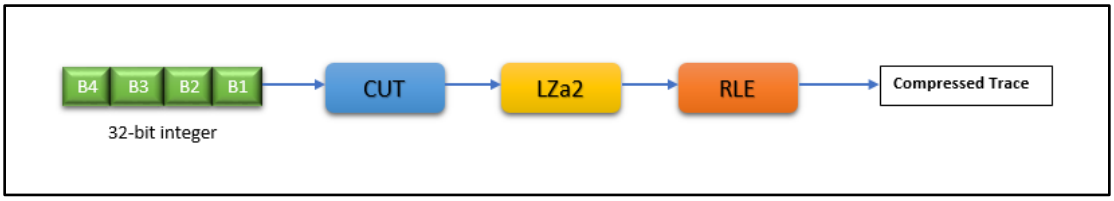

Figure 1. General information flow through the compression components

The fact that first component in our compression chain is CUT simplifies the implementation as it indicates that byte granularity suffices for our values. To implement the CUT, all the trace data of more than one byte size must be interpreted to a sequence of bytes. For example, the bitmap of active threads is a 32-bit integer which is four bytes long and must be fed to LZ component as a sequence of four bytes but the function IDs can be directly fed into LZ as they are one byte long. The output of LZ is fed into the RLE component which its output is stored into global arrays, Figure2. The final traces,

then are read from global arrays and written out. The decompression phase is done in a reverse direction from RLE up to CUT.

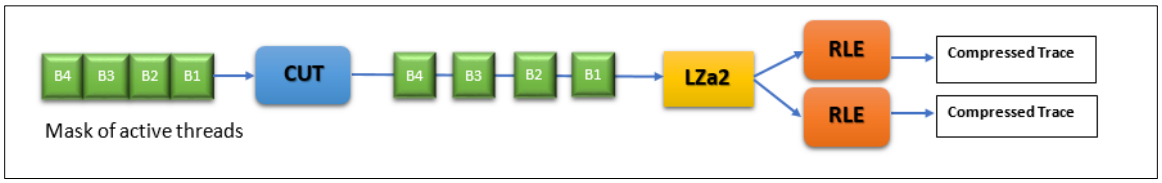

Figure 2. Diagram showing the application of CUT in the chain of compression components

A variant of the LZ77 algorithm [35] is implemented by LZan component. It incorporates tradeoffs that make it more efficient than other LZ77 versions on floating-point data and operates as follows. To identify the most recent prior occurrence of the current value in the trace location 1, a hash table is used. Afterward the n values immediately before location l is checked to see whether it matches the n values just before the current location. If they do not, the current trace value is released and the component advances to the next trace value. If the n values match, the component counts how many values following the current value match the values following location l. The length of the matching substring is emitted and the component advances by that many values. Smaller values of n yield more matches, which have the potential to improve compression, but also result in a higher chance of zero-length substrings, which hurt compression. The best algorithm reported by CRUSHER showed n = 2. Therefore, I chose LZa2 for ECL-Tracer.

The RLE component proceeds as follows. It emits a sequence of non-repeating bytes and counts how many of them there are. Then it counts how many times the last

emitted byte repeats. It records both counts in a byte. The first count gets the lower four bits and the second count the upper four bits. Because there are only four bits available, the maximum count is 15. As a consequence, it always stops counting and moves on when a count of 15 is reached so that the count doesn't go higher. I used an 18-element buffer size for RLE which is filled with trace data. The reason to choose 18 elements is, at the worst case the input sequence consists of 16 non-repeating values which all will be emitted by RLE thus the output will be 17 elements including a counter and 16 non-repeating values. This buffer is then copied to global arrays when either the buffer is full or it reaches a non-repeating value after a sequence of repeating values whichever it reaches first. From the perspective of memory consumption, the buffer takes only 18 bytes to allocate per warp, which is a reasonable amount when running many warps.

4.4 Decompression

In order to process the generated compressed trace, I needed a program to read in the compressed data so I created a decompression program. The decompressor follows the compression steps in reverse order to reproduce the original trace data.

5. EVALUATION

To evaluate and measure the performance, ECL-Tracer was tested on some LonestarGPU benchmarks as well as several algorithms with different levels of irregularity in behavior. Regular programs operate on array- and matrix-based data structures and have relatively predictable control flow and largely independent computations. In contrast, irregular programs build, traverse and update dynamic data structure such as trees, graphs, etc. Unlike regular programs, irregular codes have complex control flow, which is dependent on the input values and changes dynamically. Irregular programs are important as many important scientific programs such as data clustering, simulations, etc. are irregular. The following subsections explain regular and irregular parallel programs on which ECL-Tracer was examined.

5.1 Fractal

This code computes a fractal from the Mandelbrot Set. The Mandelbrot Set is the set of complex numbers *C* for which the sequence $z_{n+1} = z_n^2 + c$; ($z_0 = 0$) is bounded and does not diverge when iterated from $z_0 = 0$. It is called fractal because the set shows repeating patterns at every scale. The fractal code I used, based on a given maximum, calculates the number of iterations needed until $|z_k| \geq 5$, where k is the number of iterations and specifies the pixel color. The real and imaginary values of *C* is determined by the scaled x/y coordinates of each pixel. Figure 3 illustrates the picture drawn by the program. Fractals are mostly regular programs.

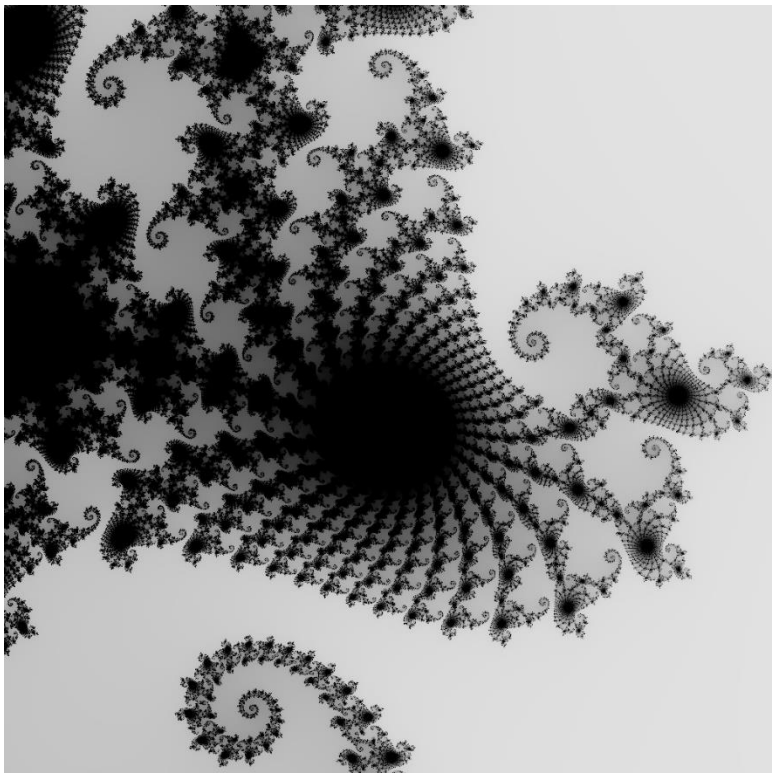

Figure 3. Fractal image from Mandelbrot Set

5.2 Maximal Independent Sets (MIS)

In a graph, a set of vertices S is called *independent* if no two vertices in the set are adjacent. This set is *maximal* if it is impossible to add another vertex to the set and still stay independent. A *maximum* independent set is a maximal independent set that has the largest set of vertices amongst maximal independent sets. Maximum independent sets are not unique. In this thesis, I used a parallel implementation of the MIS algorithm that is built based on Luby algorithm. MIS is an irregular program.

5.3 N-Body

In general, N-body simulation system consists of bodies where n represents the number of bodies. Bodies interact via pair-wise forces. N-body can be widely used as many systems can be modeled based on it such as star/galaxy clusters or particles in electric or magnetic forces. The N-body algorithm I used, gets the number of bodies and the number of time steps as inputs. In each time step, it calculates the force between bodies and updates body positions and velocities based on the calculated force. N-body falls into the category of regular programs.

5.4 LonestarGPU benchmark

In this thesis, I am using the LonestarGPU (version 2.0) [36] to generate and evaluate traces. LonestarGPU, which is created by ISS group at University of Texas at Austin in collaboration with Texas State University, is a collection of CUDA implementations of several widely-used real-world irregular applications. I chose the DMR (Delaunay Mesh Refinement) [37] and MST (Minimal Spanning Tree) [38] applications from this benchmark.

5.4.1 DMR

The DMR algorithm takes as input an unrefined triangulation of a set of points in a plane (Figure 4 (a)) [39]. The triangles that do not meet the defined criterion (angles in the mesh should not be less than 30 degrees) are called “bad” triangles (black triangles). All bad triangles are initially placed on a worklist. The refinement procedure that is

repeated in each step works as follows: 1) a bad triangle is picked from the worklist, 2) a number of triangles surrounding the bad triangle called a cavity are collected (gray triangles in Figure 4 (b)) and the cavity is retriangulated. If the retriangulation generates new bad triangles, these are added to the worklist and processed again until all bad triangles have been removed from the mesh. A mesh can be formed based on a graph in which the nodes and edges represent triangles and triangle adjacencies, respectively. I use a parallel implementation of the algorithm for this thesis.

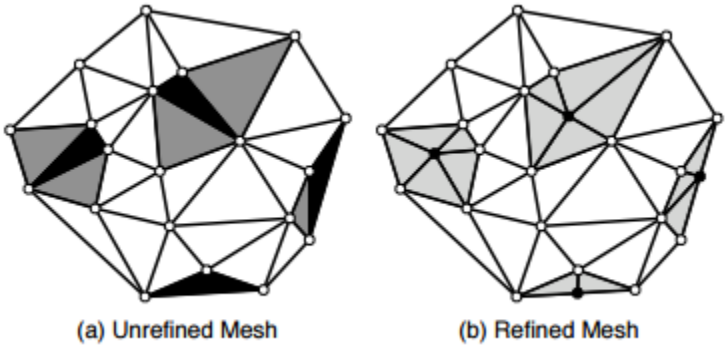

Figure 4. Delaunay Mesh Refinement

5.4.2 MST

In an edge-weighted, undirected graph in which all the vertices are connected together, a Minimum Spanning Tree (MST) is defined as the subset of the edges without any cycles and with the minimum possible total edge weight such that the edges still connect all the vertices [40]. A parallel implementation of this algorithm is used in this thesis.

5.5 Experimental Methodology and Configuration

The ECL-Tracer was developed and tested on Linux platforms. I evaluated all the programs on the ECL-Tracer and measured the runtime of the codes with and without using the ECL-Tracer. The collected runtime includes the time for running the kernel, copying the compressed trace data to the CPU and writing the data to disk.

The results are presented for two GPUs: a GeForce GTX Titan X and a Tesla K40. The Titan X is based on the Maxwell architecture with compute capability 5.2. It has 3072 processing elements, which are distributed over 24 multiprocessors and support the total of 49,152 threads. In terms of memory, each multiprocessor has 96 KB of shared memory as well as 48 KB of L1 cache. In addition, a 2 MB L2 cache and 12 GB of global memory are shared by all the multiprocessors. The default clock frequency of the processing elements and the GDDR5 global memory are 1.1 GHz and 3.5 GHz, respectively.

The Tesla K40 is based on the Kepler architecture with compute capability 3.5. The K40 comprises 2880 processing elements distributed over 15 multiprocessors that can hold up to 30,720 threads. It has 64 KB of cache that is split between the shared memory and the L1 data cache. All 15 multiprocessors share a 1.5 MB L2 cache as well as 12 GB of global memory.

Both GPUs are plugged into 16x PCIe 3.0 slots in the same system, which has dual 10-core Xeon E5-2687W v3 CPUs running at 3.1 GHz. The host operating system is CentOS 6.7 and memory size is 128 GB.

For all the applications except DMR, traces were generated on both GPUs. All codes were compiled with nvcc V8.0.61. The traces were compressed using my

compression algorithm on-the-fly (cf. Chapter 4). For comparison, I compressed the decompressed traced with Bzip2 and Gzip. Trace data were collected from experiments when running ECL-Tracer using different inputs. Table 1 lists all the inputs.

Table 1. Input Data used for Tracing.

Program Name	***Input***		
Fractal	**Width**	**Depth**	
	2048	1024	
	2048	2048	
	2048	4096	
N-Body	**Bodies**	**Time steps**	
	50000	10	
	60000	10	
MIS		**Nodes**	**Edges**
	USA-road-d.NY.egr	264346	730100
	amazon0312.egr	400727	4699738
	internet.egr	124651	387240
DMR		**Nodes**	**Triangles**
	250k.2_15	275000	524998
	r1M_12	1000003	2000001
MST		**Nodes**	**Edges**
	rmat12	4096	59320

6. RESULTS

This chapter shows the results of the experiments conducted for this thesis. The inputs were chosen from the regular and irregular programs described in Chapter 5. I investigate the overhead incurred by ECL-Tracer and the runtime of ECL-Tracer with and without compression. Then I present the comparison between the compression ratio obtained by ECL-Tracer and the compression ratio from the standard compression algorithms Bzip2 and Gzip running on Titan X. Finally, the results of conducting the experiments on the K40 are presented.

6.1 Overhead

In this subsection, I look into the overhead incurred by ECL-Tracer when tracing different applications and compressing traces on-the-fly. To obtain the overhead, I divided the runtime of the instrumented program by the original runtime of the program given the same input. Figure 5 illustrates the results in logarithmic scale. The inputs are listed along the x-axis, and the runtimes in seconds are represented on the y-axis. Numbers above 1.0 indicate a slowdown due to the tracing.

The results clearly show the low overhead of ECL-Tracer on most programs, which is below 5.5 in most cases. In particular, for irregular programs such as MIS and DMR the overhead is below 2.6, meaning that these programs run less than three times as long with tracing than they do without tracing. This demonstrates that my tracing tool does not incur an unreasonable amount of overhead on these complex programs.

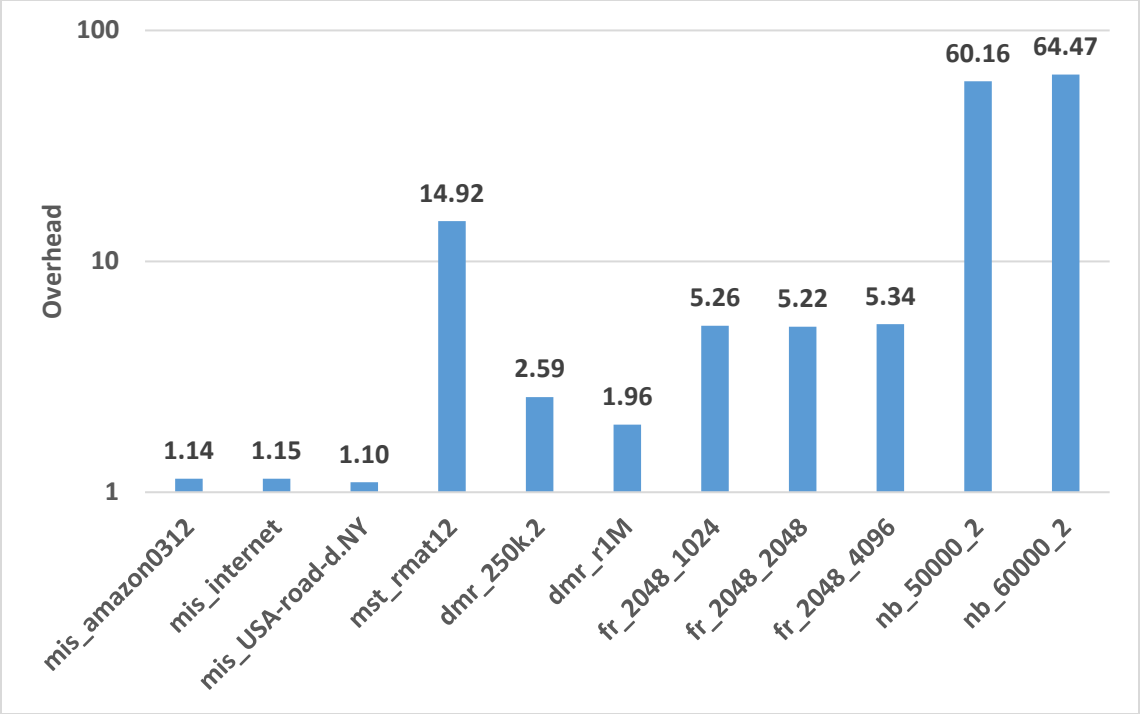

Figure 5. ECL-Tracer overhead relative to using no tracing tool

This is significant because the runtime of irregular programs is highly dependent on the input. The overhead of Fractal and N-Body, however, is relatively higher than that of other programs. This is probably because regular programs utilize the GPU hardware more effectively. Adding irregular compression code to a regular application therefore incurs a higher relative overhead than adding such code to an irregular application. Despite the fact that the overhead of the applications mentioned above is higher than for other applications, the resulting compression ratio is considerable as we will see.

6.2 Relative runtimes of ECL-Tracer with and without compression

This part analyzes the overhead of the compression technique used in the ECL-Tracer. Figure 6 show the comparison between the runtime of the ECL-Tracer with and without compression.

As shown in the figure, compressing trace data results in a runtime reduction compared to the non-compressing tracing time for the irregular programs. Significantly, the DMR runtimes decreased from 2.6s to 0.5s for the smaller input (dmr_250k.2) and from 3.8s to 1.1s for the larger input (dmr_r1m). Interestingly enough, both runtimes are roughly the same for USA-road-d.NY. Regular programs (Fractal and N-body), however, represent an increase when compressing the trace data.

My main take-away observation is that compression increases the runtime quite a bit on the regular programs but decreases it a little to a lot on the irregular programs (except on MST, which has a very short runtime). On the regular codes, the compression incurs more overhead than the savings due to having to write less data. On the irregular codes, the relative compression overhead is not that high (see above) and lower than the saving due to writing less data.

The overhead of the compression is illustrated in Figure 7. The results are computed by dividing the runtime of ECL-Tracer with compression by the runtime without compression. Numbers above 1.0 indicate that the non-compressing version of the code works faster. Interestingly enough, compression overheads below 1.0 are present for most irregular programs. Moreover, two MIS inputs and DMR run faster with trace compression, which, as mentioned earlier, is the result of writing and copying less data to disk.

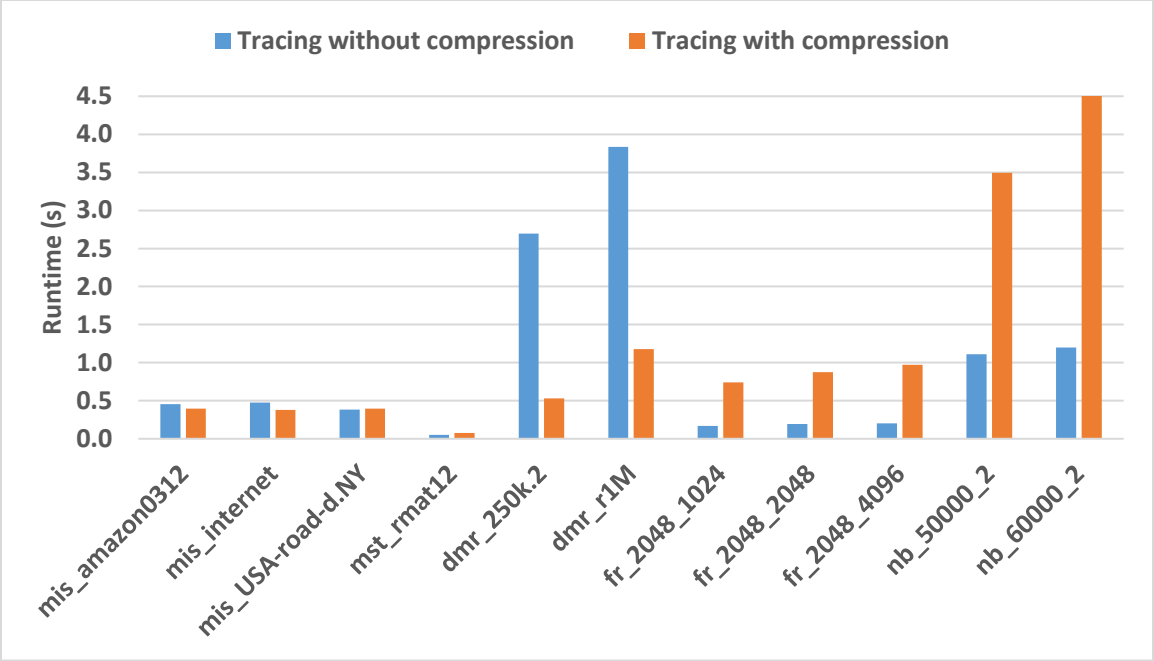

Figure 6. Runtimes of ECL-Tracer with and without compression

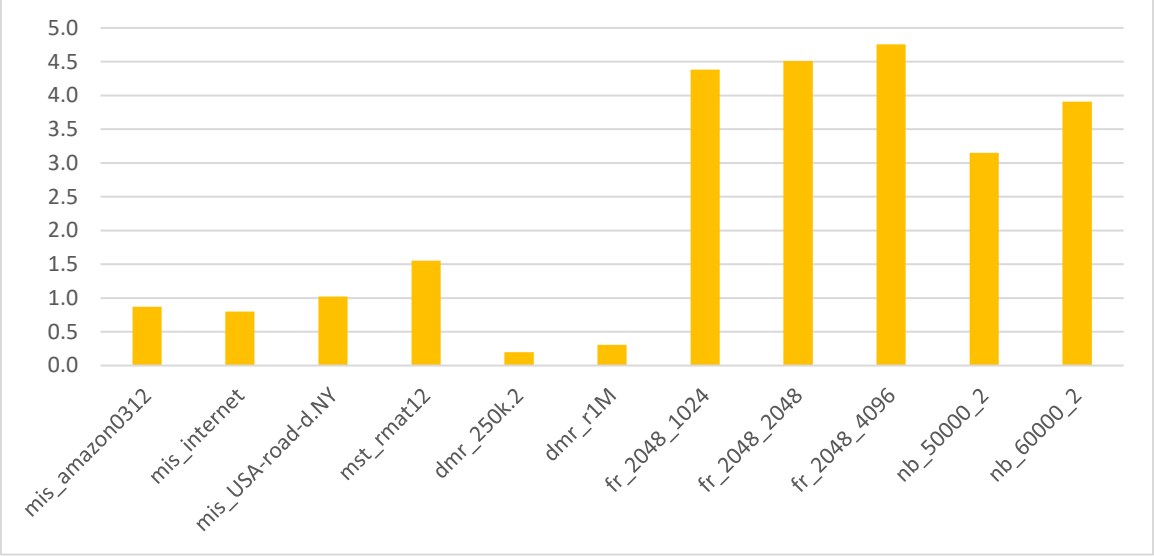

Figure 7. Overhead of compression when using ECL-Tracer

6.3 Compression Ratios

Compression ratios achieved by the compression algorithm implemented in ECL-Tracer is outlined in Figure 8. As expected, regular programs such as Fractal and N-Body yield higher compression ratios than irregular programs. The reason is simply because regular codes generate longer sequences of repeating numbers, which compress better.

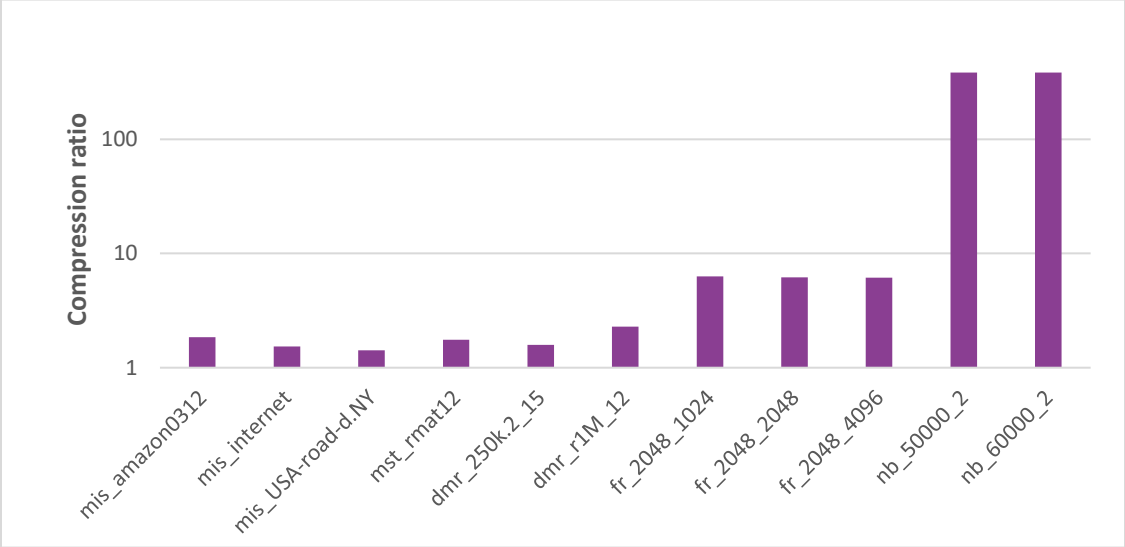

Figure 8. Compression ratio for ECL-Tracer

Figure 9 compares the compression ratio of ECL-Tracer to Bzip2 and Gzip for all the tested applications. It can be seen from Figure 9 that ECL-Tracer's compression ratio is reasonably close to Bzip2 and Gzip on most irregular programs. On average, Bzip2 compresses better than ECL-Tracer by only a factor of 2.9 and Gzip does so by factor of 3.2 on traces from irregular codes. I believe that this is a satisfactory tradeoff between overhead and compression ratio.

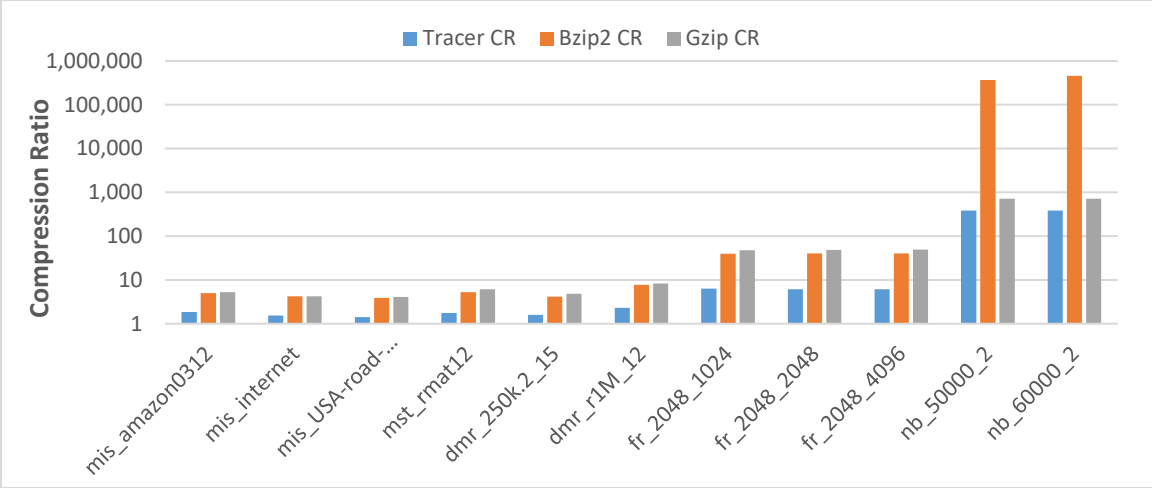

Figure 9. Compression ratios for ECL-Tracer compared to Bzip2 and Gzip

6.4 Compression speed

To compare the runtime of ECL-Tracer with two general purpose compressors, I ran the Gzip and Bzip2 on the same trace data. Figure 10 presents the runtimes of Bzip2 and Gzip relative to the runtime of ECL-Tracer. Values above 1.0 mean that ECL-Tracer runs faster than the other two tools.

The most remarkable result to emerge from the data is that Bzip2 is slower than ECL-Tracer in most cases but compresses better. Specifically, for regular applications like Fractal and N-Body, ECL-Tracer outperforms Bzip2 by a factor of 3.4 on average. This is surprising because the ECL-Tracer running time includes the entire execution time of program. The reason is that Gzip and Bzip2 are serial programs running on the CPU whereas ECL-Tracer runs in parallel as each warp in the application compresses its own trace while other warps perform their compression at the same time.

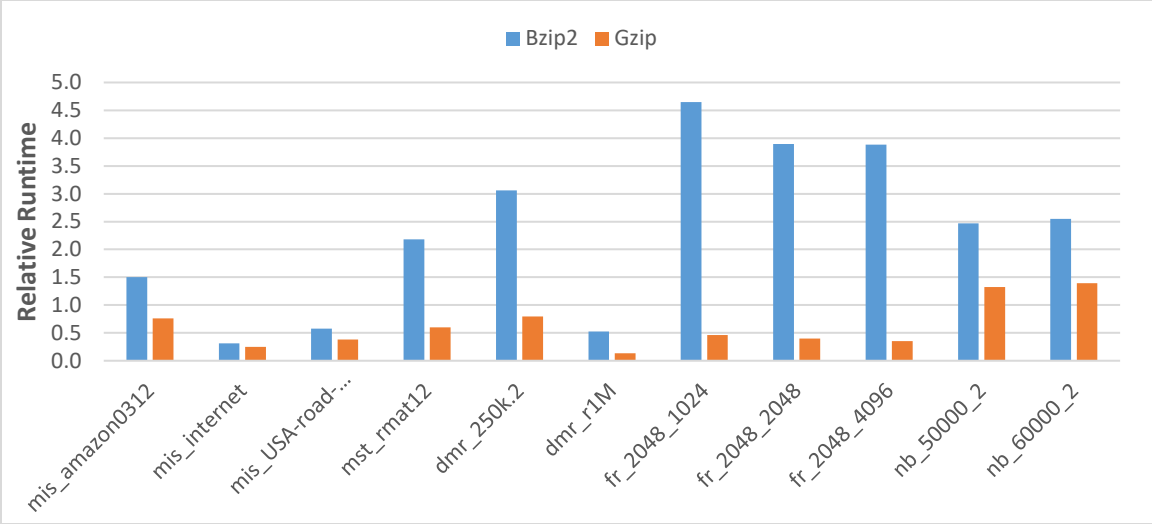

Figure 10. Runtime of Bzip2 and Gzip relative to runtime of ECL-Tracer

6.5 Results for Tesla K40

The ECL-Tracer also works on older generation of GPUs such as Tesla K40. In this subsection, I will present the result of all the experiments except DMR running on K40. DMR did not run on the K40, even without the tracing code. We observed that programs executing single-precision operations such as N-Body run slower than programs that perform the computation on double values (Fractal) on K40. The reason is Titan X by design performs better on float numbers.

I illustrate the overhead of using ECL-Tracer relative to normal execution of the applications using no tracing tool in Figure 11. As presented in the figure, the overhead on regular codes is higher. As mentioned earlier (cf. Subsection 6.1), this is probably because regular programs utilize the GPU hardware more effectively. Adding irregular compression code to a regular application incurs a higher relative overhead than adding

such code to an already irregular application. Although the overhead on regular codes is higher than on irregular programs, the resulting compression ratio is better as we will see.

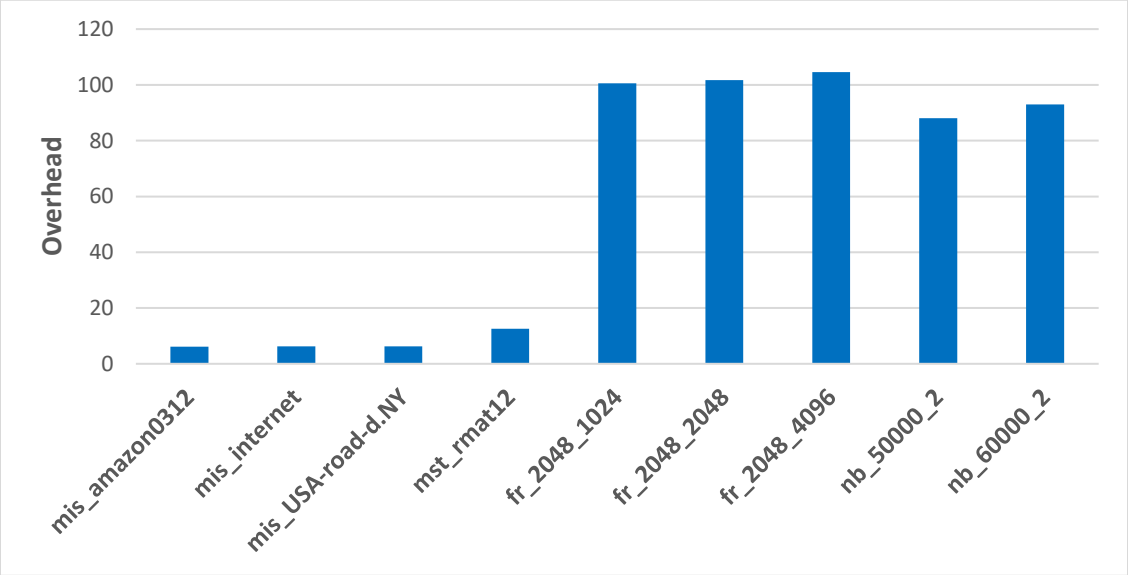

Figure 11. ECL-Tracer overhead relative to using no tracing tool on K40

The running time of ECL-Tracer with and without compression turned on is shown in Figure 12. As can be seen runtimes (except for MST) are significantly higher with compression on K40.

Figure 13 represents the compression overhead incurred by ECL-Tracer on K40. Values were computed by dividing the runtime of ECL-Tracer with compression by the runtime without compression. Numbers above 1.0 indicate that the non-compressing version of the code runs faster.

Figure 12. ECL-Tracer runtimes with and without compression on K40

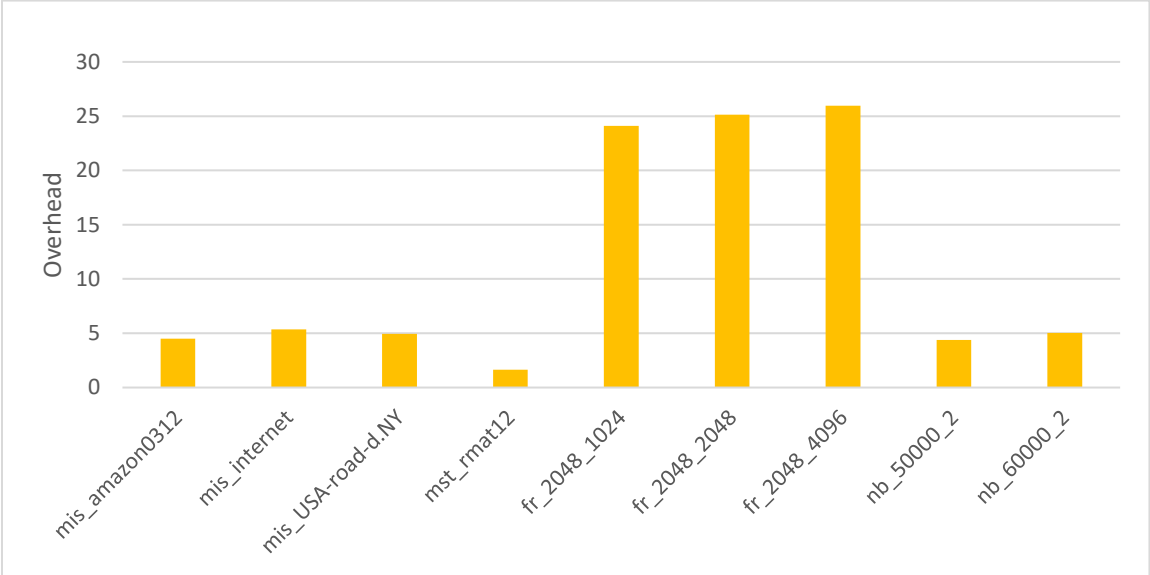

Figure 13. Compression overhead when using ECL-Tracer on K40

The figure 14 presents the compression ratio of ECL-Tracer running on K40. The results show that the ECL-Tracer compresses the trace data from regular code (Fractal and Nbody) better than the irregular codes. The reason could be the larger number of repeating numbers generated by regular codes.

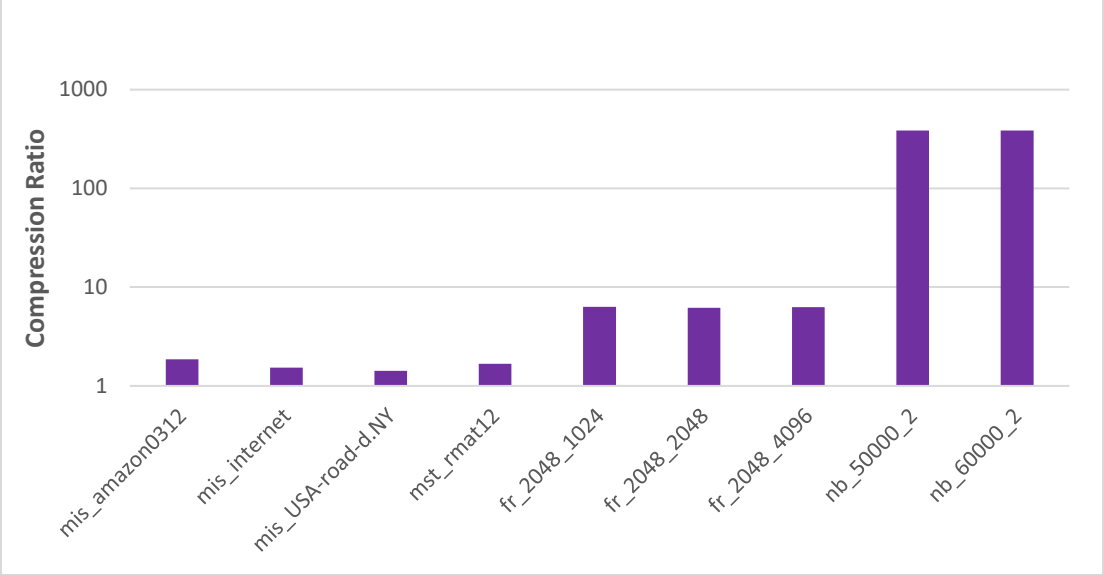

Figure 14. Compression ratios of ECL-Tracer on K40

Figure 15 compares the compression ratios of ECL-Tracer on K40 with the compression ratios achieved from compressing the decompressed files by Bzip2 and Gzip. We can see that ECL-Tracer compresses close to Bzip2 and Gzip on all the irregular codes. On average, Bzip2 compresses better than ECL-Tracer by only a factor of 2.74 and Gzip does so by a factor of 2.93 on irregular codes.

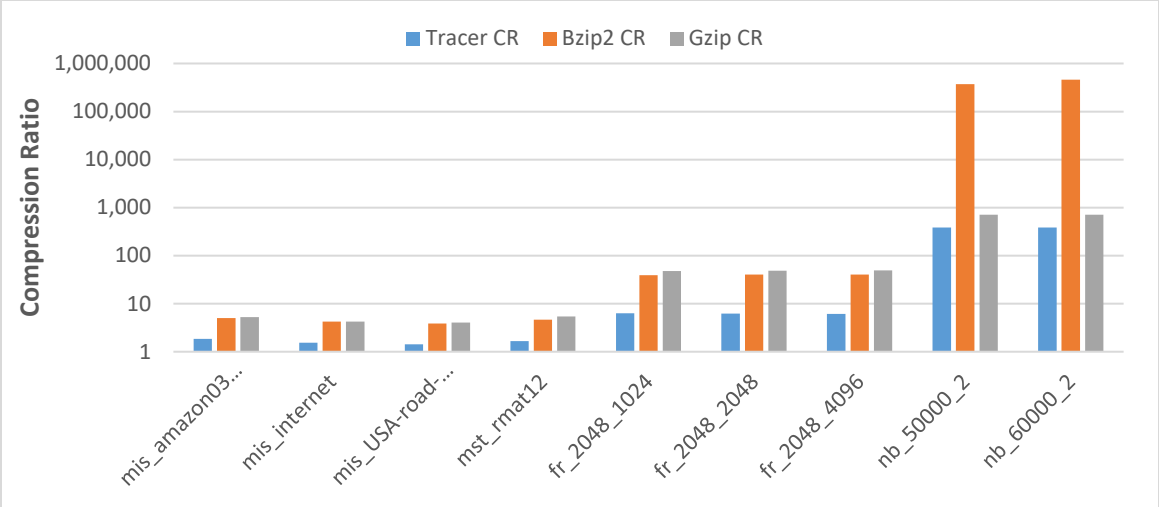

Figure 15. Compression ratios for ECL-Tracer compared to Bzip2 and Gzip on K40

Figure 16 compares the ECL-Tracer runtimes on K40 with Bzip2 and Gzip. Similar to Titan X (cf. Section 6.3, Figure 9), for most regular codes we see ECL-Tracer performs faster than Bzip2 but slower than Gzip. There is a trade-off between the running time and compression ratios in most regular code.

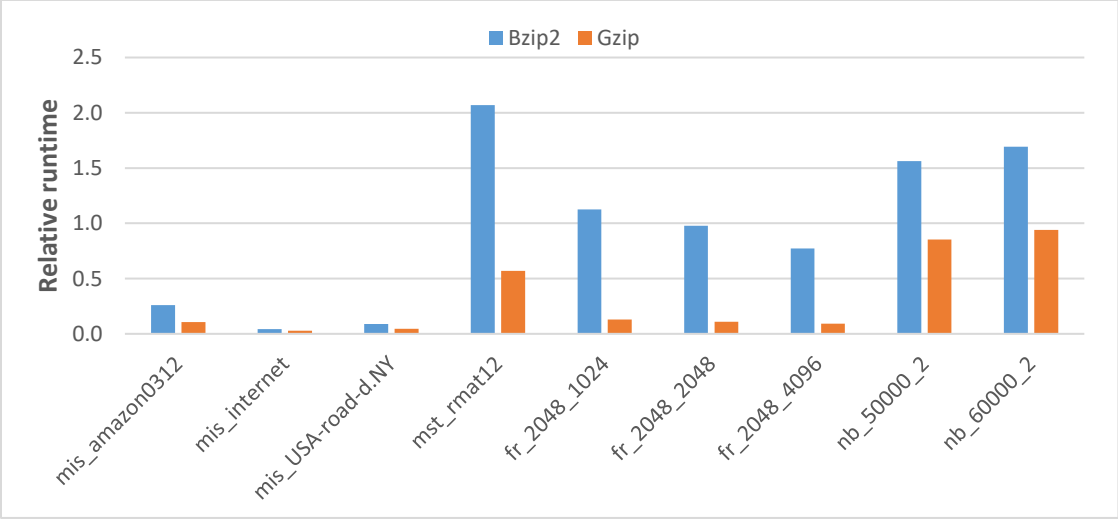

Figure 16. Runtime of Bzip2 and Gzip relative to runtime of ECL-Tracer on K40

7. CONCLUSION

7.1 Summary

This thesis presents an efficient tracing tool for massively parallel programs running on GPUs. As far as I am aware, the ECL-Tracer is the first GPU tracing tool capable of compressing the trace data on the fly before writing it to disk. This tool is a clear improvement on GPU execution tracing, and I believe that my approach would lend itself well for use by researchers and programmers to aid to increase the performance of their programs. The findings of this thesis imply that the ECL-Tracer has low overhead and is efficient in terms of time and space for most parallel programs.

7.2 Future Work

ECL-Tracer works properly on all tested CUDA programs. Nevertheless, it can easily be made even more capable to further help programmers and researchers. For example, one could add support to record additional information, which might help identify and analyze performance weaknesses or bugs faster. To expand this study, one could investigate other compression algorithms to possibly improve the compression ratio. Testing the ECL-Tracer on additional programs, different architectures, and even multi-GPU programs are also opportunities for future work.

APPENDIX SECTION

This section lists all the raw data for all the experiments.

INPUT	Decompressed file	ECL-Tracer	Bzip2	Gzip
fr_2048_1024	25511720	4043071	649107	536875
fr_2048_2048	26239135	4254701	652150	540871
fr_2048_4096	26457160	4317234	652126	541131
mis_amazon0312	4505129	2428465	902288	861001
mis_internet	773332	505281	184126	182843
mis_USA-road-d.NY	1400371	986743	360214	345481
nb_50000_2	784648909	2040629	2122	1100663
nb_60000_2	1129532409	2929635	2456	1577121
mst_rmat12	410575	233156	78016	67085
dmr_250k.2_15	6046059	3822143	1451921	1244914
dmr_r1M_12	2658209	1160153	345236	319085

Table A.1 File sizes (in Bytes) of traces generated on Titan X

INPUT	Decompressed file	Tracer	Bzip2	Gzip
fr_2048_1024	25541245	4046217	649198	536850
fr_2048_2048	26269295	4254895	652013	540869
fr_2048_4096	23019120	3677657	564250	476242
mis_amazon0312	4445099	2393580	892260	847995
mis_internet	770667	503341	182949	182447
mis_USA-road-d.NY	1399181	986043	360215	345368
mst_rmat12	364625	218161	77929	66848
nb_50000_2	784648909	2040629	2122	1100663
nb_60000_2	1129532409	2929635	2456	1577121

Table A.2 File sizes (in Bytes) of traces generated on K40

BIBLIOGRAPHY

[1] "http://www.thefreedictionary.com/trace+program," Farlex clipart collection, 2003-2008. [Online].

[2] "https://www-927.ibm.com/ibm/cas/hspc/student/tutorials/tracing.html," IBM. [Online].

[3] F. Wolf, F. Freitag, B. Mohr, M. Shirley and B. Wylie, "Large Event Traces in Parallel Performance Analysis," in *8th workshop on Parallel Systems and Algorithms*, Frankfort, Germany, 2006.

[4] W. contributors, "Instrumentation (computer programming)," Wikipedia, 2015. [Online].

[5] "https://en.wikipedia.org/wiki/Data_compression#cite_note-mahdi53-2," Wikipedia. [Online].

[6] "https://en.wikipedia.org/wiki/Gzip," wikipedia. [Online].

[7] "https://en.wikipedia.org/wiki/Huffman_coding," Wikipeia. [Online].

[8] R. Fraile, "http://blog.servergrove.com/2014/04/14/gzip-compression-works/," 14 4 2014. [Online].

[9] "http://www.gzip.org/algorithm.txt," [Online].

[10] "https://en.wikipedia.org/wiki/Run-length_encoding," Wikipedia. [Online].

[11] "https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform," Wikipedia. [Online].

[12] "https://en.wikipedia.org/wiki/Move-to-front_transform," Wikipedia. [Online].

[13] "https://en.wikipedia.org/wiki/Bzip2," wikipedia. [Online].

[14] J. Caubet, J. Gimenez, J. Labarta, L. DeRose and J. Vetter, "A dynamic tracing mecha-nism for performance analysis of OpenMP applications," Berlin, Heidelberg, Springer Berlin Heidelberg, 2001, pp. 53-67.

[15] T. E. Anderson and E. D. Lazowska, "Quartz: a tool for tuning parallel program performance," in *Measurement and modeling of computer systems*, Boulder, Colorado, 1990.

[16] L. Mallens, "A framework for data-access strategies in GPGPU programs," Eindhoven University of Technology, Eindhoven, 2013.

[17] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, 2008.

[18] "http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/," [Online].

[19] "http://www.nvidia.com/object/nsight.html," [Online].

[20] S. S. Shende and A. Malony, "THE TAU PARALLEL PERFORMANCE SYSTEM," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287-311, 2006.

[21] A. Knüpfer, H. Brunst, . J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller and W. E. Nagel, "The Vampir Performance Analysis Tool-Set," pp. 139-155, 2008.

[22] D. Couturier and M. R. Dagenais, "LTTng CLUST: A System-Wide Unified CPU and GPU Tracing Tool for OpenCL Applications," *Advances in So□ware Engineering*, 2015.

[23] S. J. Eggers, D. R. Keppel, E. J. Koldinger and H. M. Levy, "Techniques for efficient inline tracing on a shared-memory multiprocessor," in *conference on Measurement and modeling of computer systems*, Boulder, Colorado, 1990.

[24] R. H. B. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," in *workshop on Parallel and distributed debugging*, San Diego, California, 1993.

[25] R. A. Patel, Y. Zhang, J. Mak, A. Davidson and J. D. Owens, "Parallel Lossless Data Compression on the GPU," in *Innovative Parallel Computing (InPar), 2012*, 2012.

[26] A. Ozsoy, "Culzss-bit: A bit-vector algorithm for lossless data compression on gpgpus," in *International Workshop on Data Intensive Scalable Computing Systems*, 2014.

[27] M. Chłopkowski and R. Walkowiak, "A general purpose lossless data compression method for GPU," *Journal of Parallel and Distributed Computing,* vol. 75, pp. 40-52, 2015.

[28] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 Gb/s on a GPU," in *GPGPU-4 Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, Newport Beach, California, 2011.

[29] A. Yang, H. Mukka, F. Hesaaraki and M. Burtscher, "MPC: A Massively Parallel Compres-sion Algorithm for Scientific Data," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, 2015.

[30] A. Goel, A. Roychoudhury and T. Mitra, "Compactly representing parallel program executions," in *PPoPP '03 Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, San Diego, California, 2003.

[31] C.-F. Kao and I.-J. Huang, "A Cache-Based Approach for Program Address Trace Compression".

[32] M. Burtscher, "VPC3: a fast and effective trace-compression algorithm," in *SIGMETRICS '04/Performance '04 Proceedings of the joint international conference on Measurement and modeling of computer systems*, New York, 2004.

[33] A. Milenkovic and M. Milenkovic, "Stream-Based Trace Compression," *IEEE Computer Architecture Letters*, vol. 2, no. 1, pp. 4-4, 2006.

[34] "http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4UzzuLA7b," NVIDIA. [Online].

[35] "https://en.wikipedia.org/wiki/LZ77_and_LZ78," 7 December 2016. [Online].

[36] U. o. T. A. Austin, "http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu," 2014. [Online].

[37] L. P. Chew, "Guaranteed-quality mesh generation for curved," *SCG*, 1993.

[38] D. Eppstein, "Spanning Trees and Spanners," *Handbook of Computational Geometry*, p. 425–461, 1999.

[39] M. Kulkarni, M. Burtscher, C. Caşcaval and K. Pingali, "Lonestar: A Suite of Parallel Irregular Programs".

[40] "https://en.wikipedia.org/wiki/Minimum_spanning_tree," Wikipedia, 1 March 2017. [Online].

[41] "https://msdn.microsoft.com/en-us/library/aa983649(VS.71).aspx," Microsoft. [Online].

[42] "http://www.gzip.org/," 2003. [Online].