LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-236179

Storage-Intensive Supercomputing Benchmark Study

J. Cohen, D. Dossa, M. Gokhale, D. Hysom, J. May, R. Pearce, A. Yoo

November 5, 2007

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Storage-Intensive Supercomputing Benchmark Study

Jon Cohen Don Dossa Maya Gokhale David Hysom John May
Roger Pearce Andy Yoo

October, 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Critical data science applications requiring frequent access to storage perform poorly on today's computing architectures. This project addresses efficient computation of data-intensive problems in national security and basic science by exploring, advancing, and applying a new form of computing called storage-intensive supercomputing (SISC). Our goal is to enable applications that simply cannot run on current systems, and, for a broad range of data-intensive problems, to deliver an order of magnitude improvement in price/performance over today's data-intensive architectures.

This technical report documents much of the work done under LDRD 07-ERD-063 Storage Intensive Supercomputing during the period 05/07-09/07. The following chapters describe

- a new file I/O monitoring tool *iotrace* developed to capture the dynamic I/O profiles of Linux processes
- an out-of-core graph benchmark for level-set expansion of scale-free graphs
- an entity extraction benchmark consisting of a pipeline of eight components
- an image resampling benchmark drawn from the SWarp program in the LSST data processing pipeline.

The performance of the graph and entity extraction benchmarks was measured in three different scenarios: data sets residing on the NFS file server and accessed over the network; data sets stored on local disk; and data sets stored on the Fusion I/O parallel NAND Flash array. The image resampling benchmark compared performance of software-only to GPU-accelerated.

In addition to the work reported here, an additional text processing application was developed that used an FPGA to accelerate n-gram profiling for language classification. The n-gram application will be presented at SC07 at the High Performance Reconfigurable Computing Technologies and Applications Workshop.

The graph and entity extraction benchmarks were run on a Supermicro server housing the NAND Flash 40GB parallel disk array, the Fusion-io. The Fusion system specs are as follows: SuperMicro X7DBE Xeon Dual Socket Blackford Server Motherboard; 2 Intel Xeon Dual-Core 2.66 GHz processors; 1 GB DDR2 PC2-5300 RAM (2 x 512); 80GB Hard Drive (Seagate SATA II Barracuda). The Fusion board is presently capable of 4X in a PCIe slot.

The image resampling benchmark was run on a dual Xeon workstation with NVIDIA graphics card (see Chapter 5 for full specification). An XtremeData Opteron+FPGA was used for the language classification application.

We observed that these benchmarks are not uniformly I/O intensive. The only benchmark that showed greater that 50% of the time in I/O was the graph algorithm when it accessed data files over NFS. When local disk was used, the graph benchmark spent at most 40% of its time in I/O. The other benchmarks were CPU dominated. The image resampling benchmark and language classification showed order of magnitude speedup over software by using co-processor technology to offload the CPU-intensive kernels.

Our experiments to date suggest that emerging hardware technologies offer significant benefit to boosting the performance of data-intensive algorithms. Using GPU and FPGA co-processors, we were able to improve

performance by more than an order of magnitude on the benchmark algorithms, eliminating the processor bottleneck of CPU-bound tasks. Experiments with a prototype solid state nonvolative memory available today show 10X better throughput on random reads than disk, with a 2X speedup on a graph processing benchmark when compared to the use of local SATA disk.

The SISC team in May-Oct 2007 was as follows: Jon Cohen, Lisa Corsetti, Don Dossa, Maya Gokhale, Eric Greenwade, John Grosh, David Hysom, Arpith Jacob, John Johnson, Scott Kohn, Vijay Kumar, John May, Marcus Miller, Roger Pearce, Craig Ulmer (Sandia CA), Andy Yoo.

Chapter 2

IOtrace Tool

2.1 Tracing I/O Behavior

An important aspect of understanding I/O performance is knowing how different applications interact with the file system. To monitor these interactions, we have developed a library called **iotrace** that records the details of each call to any of a selection of low- and high-level I/O routines, such as **open, close, read, write, fopen,** and **fscanf**. Our library requires no changes to the monitored application; the user simply interposes **iotrace** between the application and the dynamically-linked I/O libraries using standard Linux techniques. (Similar techniques are available under other operating systems, including AIX, but we have not yet ported **iotrace** to these systems.)

The **iotrace** library can store trace data in a text format that can be imported into a spreadsheet or the PerfTrack [11] performance database. It can also store data in a more compact binary format. We have written a companion program that reads binary files and produces a series of graphical summaries of the I/O traces.

This chapter describes the usage, design, and limitations of **iotrace**, and it notes areas for further development.

2.1.1 Using iotrace

Once **iotrace** has been built and installed (as described in the documentation that accompanies it), an application's I/O activity can be monitored as follows:

```
env LD_PRELOAD=/full/path/iotrace.so myapp args...
```

The **env** command runs the specified application (in this case, **myapp**) with one or more specified additional environment variables set. **LD_PRELOAD** is a standard Linux variable that causes the named library to be loaded dynamically before any of the application's other dynamic libraries. As a result, the **open, close, read, write** and other functions in **iotrace.so** are called instead of the standard versions. The **iotrace** versions of these functions forward the calls to their standard counterparts and then record trace data for each call.

The example above shows how to apply **iotrace** to a single command. If the user sets **LD_PRELOAD** to point to **iotrace.so** in the shell environment, then all subsequent commands will use have their I/O activity monitored. This should work correctly, and by default (see below) **iotrace** will create a separate log file for each process, but the amount of data generated in this way could be overwhelming.

Currently, **iotrace** monitors the following functions:

open	**close**	**read**	**write**
fopen	**creat**	**fopen64**	**open64**
fscanf	**fread**	**fwrite**	**fclose**

The technique for monitoring calls is generic, so additional functions could be included relatively easily. I/O operations on **stdin, stdout, stderr** and on the log file itself are not logged.

Environment variables

Several environment variables are available to control the behavior of **iotrace**. These may be set as part of the **env** command, or they may be set in the shell environment.

IOTRACE_APPNAME This variable tells **iotrace** the name of the application being monitored. (The library has no access to the command line, so it cannot determine this automatically.) This information can be used in two ways: as part of the name of the log file, and in the output data itself. If the variable is not set, **iotrace** uses the default name **unknown**. See the description of the IOTRACE_LOGFILE environment variable, below, for a complete description of how the log file name is set.

IOTRACE_BINARY Set this variable to the word **true** to output the log file in a more compact (but less portable) binary format. If the variable is not set or if it is set to any other value, the log file is written in text format.

IOTRACE_EXECNAME This variable, if set, should contain a string that uniquely identifies a particular execution of the target application. For example, it could contain the application name followed by a process identifier or a time stamp. As with IOTRACE_APPNAME, this text can be used both in the log file name and in the data output. If the text contains either `%d` or `%i`, **iotrace** will substitute the process identifier of the application for this format string. (No more than one format command should appear in the name, and since **iotrace** passes this text directly to **snprintf**, using any other format command in the text could produce gibberish.) If this variable is not set, the execution name defaults to ***appname.pid***, where ***appname*** is the value of IOTRACE_APPNAME (or its default), and ***pid*** is the process identifier of the running application.

IOTRACE_LOGFILE This variable contains the name to use for the log file. As with the IOTRACE_EXECNAME, it may contain a `%d` or `%i` format command, with the same functionality and caveats. If output is in binary form, a file named as specified will contain the main output, and a second file with the suffix **.summary** will contain additional (ASCII-formatted) summary data. For text-format output, there is no summary file. If this variable is not set, then the name of the log file file is ***execname*_log.tsv** for text-formatted output and ***execname*_log.bin** and ***execname*_log.bin.summary** for binary output. Given the rules stated above for defaults, if no environment variables are set, **iotrace** will write the log file in text format, and the file will be called **unknown.*pid*_log.tsv**.

Output formats

A text-format log file consists of a header line listing the names of the columns followed by rows of tab-separated values, one for each I/O event that was monitored. Many spreadsheet and graphics programs, as well as the PerfTrack performance database, can read this file format. Table 2.1 lists the columns in the log file.

The **iotrace** library will normally create a separate log file for each process, so a parallel job will create one log per task. Multithreaded processes will create a single file. When a single job uses several processes, it important to ensure that each process uses a different file, since **iotrace** will not reopen a log file that already exists.

The binary format is designed to save disk space and I/O overhead for applications that write large volumes of data. The disadvantage of this format is that it is not portable between architectures (for example, between 32- and 64-bit machines, or big-endian and little-endian machines.) Also, the file can only be read by the **iotrace** data parsing tool and cannot be imported into other applications.

The binary output is written in two files, one with the suffix **.summary** and the other **.bin**. The **.summary** file maintains general summary information in text format about the overall process and every

Column name	**Contents**
Application	Application name, as specified in IOTRACE_APPNAME, or its default
Execution	Execution name, as specified in IOTRACE_EXECNAME, or its default
Tool	Always "IOTRACE"
Filename	Name of file on which operation was done, as specified in the call that opened it
fd	Unix file descriptor for the file
Op sequence	Sequence number for this operation, starting with 0 for the first I/O call in the application. (This number guarantees that every line in the log will be unique.)
Operation	Name of the I/O call
Bytes	Actual number of bytes transferred in the operation, or 0 for calls (such as **open**) that transfer no data, or -1 if the call resulted in an error
start_time	System clock time (in seconds) when operation began
end_time	System clock time (in seconds) when operation ended
Time (sec)	Difference between start_time and end_time
PID	Process ID for this execution

Table 2.1: Output columns for the **iotrace** text-format log file.

Data Field	Contents
App_Name	Application name, as specified in IOTRACE_APPNAME, or its default
Exe_Name	Execution name, as specified in IOTRACE_EXECNAME, or its default
Tool_Name	Always "IOTRACE"
Binary_Trace_File	Filename that holds the **.bin** trace output
Total_Num_IO_events	The total number of I/O events captured
Total_IO_bytes	The total number of bytes moved during execution
Total_IO_time	The total time spent in I/O routines
Start_Time	System clock time (in seconds) marking the start of process execution
End_Time	System clock time (in seconds) marking the end of process execution
Number_of_Files	Total number of files used during execution
File Level statistics, see Table 2.3	

Table 2.2: Output data in the **iotrace .summary** file.

Column Name	Contents
Filename	Name of file on which operation was done, as specified in the call that opened it
Total Number Read	Total number of read operations for a given file
Total Bytes Read	Total number of Bytes read for a given file
Total Time Read	Total amount of time reading a given file
Total Number Write	Total number of write operations to a given file
Total Bytes Write	Total number of Bytes written to a given file
Total Time Write	Total amount of time spent writing to a given file
Global File ID	Unique mapping for this file used with the **.bin** file
fd	Unix file descriptor for the file

Table 2.3: File-level summary data in the **iotrace .summary** file

file handle used throughout execution. An overview of the **.summary** can be found in Tables 2.2 and 2.3. Among other things, this file is responsible for maintaining a unique file mapping that is used with the **.bin** trace data. The **.bin** file holds a sequence of binary records described in Table 2.4, the data width of each record may vary on different platforms.

Tracing Java programs

The **iotrace** library can record the I/O behavior of Java programs simply by including it in the Java command line:

```
javac Myclass.java
env LD_PRELOAD=iotrace.so java Myclass
```

The library will capture the I/O events from both the Java program and the Java Virtual Machine, but since the log file will record the file name for each operation, it should be easy to separate the application I/O from Java's.

Data Field	Contents
unsigned long long int bytes	Number of Bytes in this I/O event
unsigned long long int file_offset	Byte offset into file before I/O event
double start_time	System clock time (in seconds) when operation began
double end_time	System clock time (in seconds) when operation ended
pid_t pid	Process ID for this execution
io_operation io_op	An enumeration value representing the type of operation (open, close, read, write, etc.)
int gfid	Global File Id, mapped using the **.summary** file

Table 2.4: I/O event record data in the **iotrace .bin** file

Plotting data from binary files

A C++ tool has been developed to parse and plot the **.summary** and **.bin iotrace** output. Four plot formats are currently supported:

Plot Type	Description
Accumulated non-IO time	Plot of accumulated time between I/O calls vs. total process running time
Accumulated IO time	Plot of accumulated Time in read/write vs. total process running time
Accumulated Bytes Moved	Plot of accumulated Bytes moved in read/writes vs. total process running time
Time-slice Bytes moved	Plot of Bytes in I/O per time-slice

2.1.2 Design of iotrace

The **iotrace** library uses a well-established technique for intercepting library calls in Linux, namely the LD_PRELOAD environment variable. When Linux loads an application with dynamically linked libraries, it first loads libraries listed in LD_PRELOAD. This means that when the application make calls to standard library functions, it will look for them first in the preloaded libraries. Our library includes functions that match the signatures of the standard I/O functions listed earlier, so calls to these functions go to our library. Each of the listed functions in **iotrace** first does some initialization (if no function in our library has been called previously) and then records the current system time before calling the original version of the corresponding function to complete the user's request. After the call returns, the **iotrace** function records the system time and the return value of the underlying call, and it stores this data in the log file. To find the underlying version of each system call that it intercepts, **iotrace** uses the **dlsym** function to look up the next version of the function in the list of the dynamically loaded libraries. This search need only be done once per function each time the monitored application runs.

All of this depends on the application being dynamically linked, which is the default in Linux. We have not experimented with using **iotrace** on statically linked applications.

Some programs fork and create child processes. Usually, the child processes inherit the parent's environment variables, so they too will see the LD_PRELOAD variable and load the **iotrace** library. Since a child process initially inherits its parent's entire address space, it will also contain a copy of the **iotrace** data structures and open file descriptors from the parent. Therefore, if a child process does I/O before it calls **exec** to load a new executable, these operations will be recorded in the parent's log file. However, once the

child loads a new executable, it will reload the **iotrace** library with independent copies of its data structures and initialize a new log file. (There is a hazard that if the parent and child do concurrent I/O operations before the child calls **exec**, then their simultaneous access to the log file could corrupt it. In practice, child processes rarely read and write data before they call **exec**.)

The **iotrace** library supports multithreaded programs by placing locks around accesses to its internal data structures. The library must be compiled with thread support for this to work. This allows nonthreaded programs to link an nonthreaded version of **iotrace**, which is slightly more efficient and which does not require the pthreads library to be available.

2.1.3 Limitations

Because **iotrace** is built on the Linux LD_PRELOAD mechanism, it is subject to certain limitations. Some higher-level I/O operations in the C Standard Library call lower level functions, such as **read** and **write**. However, since these high-level and low-level functions all reside in the same library, **iotrace** cannot intercept calls between them. For example, when **fscanf** calls **read** to input its data, **iotrace** cannot intercept this intra-library call. Instead, **iotrace** can only include **fscanf** among its intercepted calls.

Conversely, other Standard Library functions' I/O operations cannot be traced because they use lower-level mechanisms for reading and writing data. The best example of this is the **mmap** call, which lets users specify a file as the backing store for an out-of-core data structure. The operating system uses its internal page fault mechanism to support this functionality, which operates below the level of the Standard Library's **read** and **write** calls. As a result, **iotrace** cannot monitor the paging activity that **mmap** generates.

Our library is designed primarily to monitor the pattern of I/O *requests* that an application generates, and only to a lesser extent the file system's *response* to them. Although we have attempted to make the monitoring of each I/O call as efficient as possible, some overhead is inevitable, and since trace data is recorded to the file system (which may or may not be the same one that the application is using), **iotrace** may compete with the application for storage resources. However, in most cases, this interference does not affect the number, size, or type of I/O requests that an application issues. The main area of potential interference is in the timing of I/O requests. We exclude much of the **iotrace** overhead from the reported I/O times, and what remains should be small compared to the cost of most I/O operations. However, it's conceivable that an application might make many short I/O requests, and that **iotrace** could introduce significant delays at the beginning and end of each of these calls. This overhead would appear to be part of the compute time of the application (rather than the I/O time), so the overall ratio of compute time to I/O time, and the time interval between individual calls, could be distorted when **iotrace** is in use.

2.1.4 Future work

The **iotrace** library could be extended in a number of ways, depending on user needs. The most simple extension is to increase the number of I/O calls that it monitors. Another easy modification would be to allow users to insert tags into to log file by calling a specified function from the application. These tags would allow the user to record application events of interest so that the log file could be interpreted more easily. Of course, making direct calls to **iotrace** would mean changing the user's source code, which is not currently required. It would also mean that the application source would need to include an **iotrace** header file to declare the logging function and then link **iotrace** directly into the executable, rather than using the LD_PRELOAD mechanism.

While the binary file format reduces the size of trace files somewhat, applications can still generate many gigabytes of trace data, which may quickly become unmanageable. Instead of recording individual data on each I/O event, **iotrace** could manage an ongoing statistical summary of events, which would remain in memory until the application finished executing. This summary could be in the form of a histogram or a set of statistics about the I/O operations seen. This approach would require substantially less storage than tracing, but it would also limit the types of analysis that could be done on the data once it had been generated. Maintaining statistics in real time might also increase the monitoring overhead modestly.

However, the cost of the additional computation would probably be small compared to the time saved by not writing full trace data to disk.

Chapter 3

An External Memory Benchmark for Large Semantic Graph Analysis

3.1 Description of Graph Benchmark

Many graph applications are highly memory-intensive, while their computation and communication demand is very low [14]. The memory-intensive nature of graph applications has an even greater impact on the performance when the graph is stored in and accessed from external storage. There are particular issues that one needs to consider when developing out-of-core (OOC) graph applications. First, how the graph data is represented and stored has significant impact on performance due to the characteristics of external disk storage devices: long latency and block I/O. Another important issue is the construction of an indexing structure to efficiently locate graph elements. Finally, the graph application must carefully arrange the order it accesses graph elements. We have developed a benchmark to capture the computational and memory access behavior of typical external memory graph applications with these issues in mind. The graph benchmark ingests a graph into the external storage and then searches the ingested graph. A detailed description of the benchmark follows.

3.1.1 Underlying data management system

The benchmark relies on SGRACE (Streaming GRAph Clustering Engine), a graph data management system designed to support the efficient storage and retrieval of large streaming graphs and to manage and access the graph. The SGRACE stores graphs in such a way that the number of I/Os is minimized and enables users to quickly explore the structure of the graphs. Figure 3.1 depicts the SGRACE architecture.

The SGRACE consists of two components based on flat files, the *partition index files* and the *partition files*. These two sets of files are used to store adjacency lists that represent the given graph. A set of vertices that are adjacent to a source vertex are stored in a partition file (PARTITION). The location of a group of adjacent vertices in PARTITION is kept in an entry in PIF, and each entry is indexed by the vertex ID. In order to retrieve all the vertices adjacent to v, we first retrieve the location, l, of the adjacent list from a PIF entry using the ID of v, and then retrieve the adjacent vertices from PIF, located by l. It should be noted that the adjacency lists are not necessarily stored in contiguous locations in a partition file, but rather in non-contiguous set of chunks chained together as shown in Figure 3.1, because it is unlikely that all the adjacent vertices arrive at the system at the same time. Retrieving a complete adjacency list, therefore, usually requires several pointer-chasings.

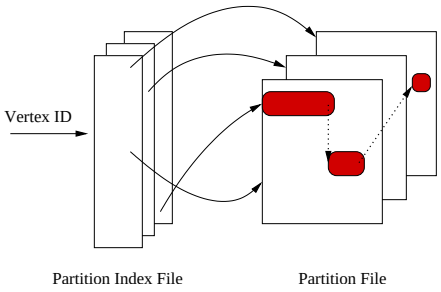


Figure 3.1: The architecture of SGRACE graph data management system

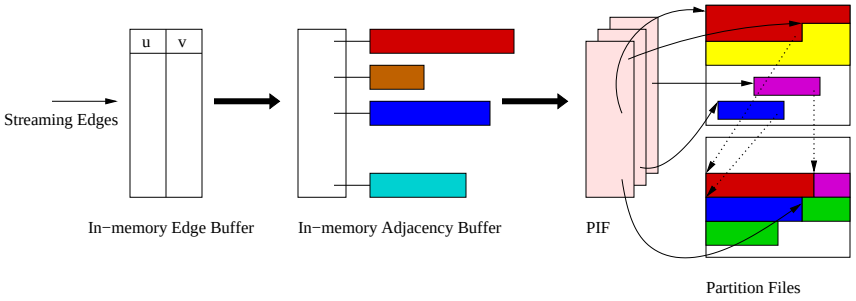


Figure 3.2: Ingestion of streaming graph

3.1.2 Graph benchmark

Typical OOC graph applications are usually optimized to reduce the number of disk IO operations, since each operation involves significant overhead in terms of latency. To reduce the number of disk accesses, the use of main memory should be maximized and accesses to disk should be performed in batch. Furthermore, the data is brought into the memory by blocks, and the disk accesses are sequentialized to reduce the latency. The benchmark models this behavior of OOC applications.

Graph ingestion

The graph ingestion phase of the benchmark reads streaming edges of a graph, builds a subgraph that consists of the new edges, and then writes the constructed subgraph to the storage. The ingestion code uses two in-memory buffers, *edge buffer* and *adjacency buffer*, to store incoming edges and a subgraph that is comprised of the edges read, respectively.

Figure 3.2 illustrates the ingestion process. First, incoming edges are read and stored in the edge buffer. When the edge buffer becomes full, all the edges in the edge buffer are merged into the subgraph being constructed in the adjacency buffer. The subgraph in the adjacency buffer is maintained as a set of adjacency lists. When an edge is merged, it is added to a group of vertices in an adjacency list that share the same source vertex. Given an edge, the adjacency list to which the edge should be merged is located by using an in-memory hash table, indexed by the source vertex of the edge.

When the adjacency buffer becomes full, the adjacency lists in the buffer are written to the external storage. The adjacency lists are sorted first by the source vertex IDs. This is to access PIF files in a sequential order. Given an adjacency list, an entry in the PIF file for its source vertex is read to check if the vertex and its adjacent vertices have been written to the storage previously. If the vertex is new, the location in the partition file where its adjacent vertices will be stored is determined and recorded in the PIF file entry. The location in the partition file is also recorded in a separate array for the writes to be performed later. If the source vertex is already in the storage, then the location of the last existing block that contains adjacent vertices is recorded in the array and the PIF entry is updated accordingly. Actual writes to the

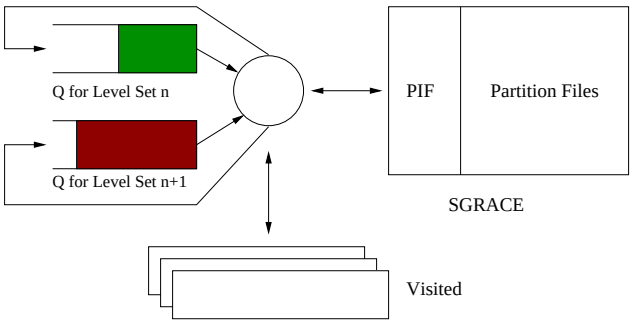


Figure 3.3: Level set expansion operation

storage are performed by reading the information recorded in the array, which is sorted first to sequentialize the accesses. These writes are all append operations. Newly appended blocks are linked to an existing block if necessary, which requires additional accesses to the partition files.

3.1.3 Graph search

The search portion of the benchmark performs a well-known graph search operation called *level set expansion*, which is an extension of classic breadth-first search (BFS). In the level set expansion, search starts from an arbitrary vertex[1]. A set of vertices called *level set* is built at each level. The initial frontier consists of a single vertex, which is the starting vertex. A level set is expanded to the next level by constructing a new level set (for the next level) consisting of vertices that are adjacent to the vertices in the current level set. Any vertices that have been visited previously are excluded from the new level set. This process of expanding level sets continues until all the vertices in the graph are visited.

Just as in BFS, two data structures are used: two queues to hold the vertices in the current and next level sets (Qs) and a bit-map to mark visited vertices (VI). In the benchmark, these two data structures are implemented as semi-external data structures. A semi-external data structure refers to a data structure in which part of its data is stored externally. When these data structures are accessed, the accesses are performed in batch in a sequential order as in the ingestion.

First, a portion of the current level set is read from a queue containing the level set to a memory buffer. Any vertices that have already been visited are then filtered out by checking V. For the remaining vertices in the buffer, the search code marks them as visited in V and collects the adjacent vertices from the PIF and PARTITION files. The collected vertices are stored in another queue that represents the next level set. This repeats until both queues become empty. The level set expansion operation is illustrated in Figure 3.3.

3.2 Performance Evaluation

We ran the graph benchmark to profile its I/O behavior on the Fusion-io server. Its performance was measured using *iotrace* discussed in Chapter 2.

The graph data is accesses from three different storage devices: Fusion-io NAND Flash array on PCIe, local SATA disk, and on a NFS-mounted RAID. These storage devices differ from each other in terms of latency, communication involved, and asynchrony in read/write speeds.

The performance was measured with various input parameters. A wide range of different graphs are used including real graphs such as Kompass [9] and IMDB [13] and a set of synthetically generated scale-free graphs [7]. Different block sizes are used, where a block is a unit of read and write operation. Block read/write usually reduces the number of I/O operations if the application has good locality.

[1] In the benchmark the starting vertex is always vertex 0.

Graphs	Op.	Dev.	Execution Time (Sec) (I/O %)					
			256B	512B	1 KB	2 KB	4 KB	8 KB
IMDB	I	N	104.9 (30)	100.1 (31)	95.3 (28)	95.7 (30)	102 (35)	103.5 (36)
		L	74.8 (5.4)	70.5 (4.6)	69 (4.6)	67.3 (4.1)	67 (4.2)	67.3 (4.3)
		F	77 (6.6)	72 (5.6)	69 (4.9)	68.7 (4.8)	68.7 (5.7)	68.6 (5.9)
	S	N	122 (37)	108.8 (36)	98.3 (33)	93.4 (32)	93.1 (35)	91.5 (38)
		L	134 (46)	127.8 (49)	118.3 (49)	119.6 (51)	116.4 (50)	115.1 (50)
		F	96.7 (25.6)	85.8 (25)	81.3 (25)	78 (24.7)	75.8 (24.8)	74.4 (24.5)
DBLP	I	N	54.9 (66)	47.8 (71)	41 (69)	36.3 (62)	49 (70)	47 (76)
		L	16.1 (10)	12.4 (4.4)	11.6 (4.7)	11.3 (3.5)	11.1 (3.3)	11.1 (3.7)
		F	13.8 (7.5)	12.3 (5.6)	11.6 (4.6)	11.3 (4.1)	11.4 (5.2)	11.7 (5.3)
	S	N	65.3 (51)	28.6 (50)	63 (55)	56.2 (33.8)	58.7 (33.7)	53.3 (47.3)
		L	46.8 (35)	43.5 (37)	42.2 (37)	44.7 (39)	42.4 (34)	38 (38)
		F	23.6 (28)	21.9 (27.7)	21.1 (28)	20.8 (28)	20.5 (28)	20.3 (27)
$S_{45,1M,5}$	I	N	72.3 (27)	63 (23.4)	63 (25)	60.8 (23)	64 (28)	67 (31)
		L	54 (7.2)	54 (12.3)	49 (5.8)	48 (5.7)	48 (5)	50 (8.7)
		F	55.5 (6.3)	52 (4.8)	48.7 (4.2)	48 (4)	49.4 (4.4)	47.7 (4.8)
	S	N	163.3 (38)	133 (41)	100 (31)	85 (26)	83 (30)	82 (31)
		L	139 (35)	141 (50)	103 (40)	95.3 (40)	94.5 (41)	91.5 (40)
		F	125 (22.4)	101 (21)	89.6 (20)	79 (20)	82.5 (18.7)	80 (18)

Table 3.1: Comparison of the graph benchmark performance for varying parameters. Symbols N, L, and F denote the network storage, local disk, and Fusion-io board, respectively. The total execution time presented includes the I/O time. A graph $S_{I,V,D}$ denotes a synthetic graph with V vertices and average degree of D, where there I high-degree vertices called *hubs*. The internal edge and adjacency buffers used in the experiments have 2 million and 1 million entries respectively.

Table 3.1 compares the ingestion and search performance of the graph benchmark for different graphs and block sizes. Two real graphs, IMDB [13] and DBLP [10], and a synthetic graph with 1 million vertices and the average degree of 5 was used in the experiment. The input data is read from the same storage device where PIF and PARTITION files are stored. The total execution time and I/O time are measured. In Table 3.1, the I/O time is given as the percentage of the total execution time.

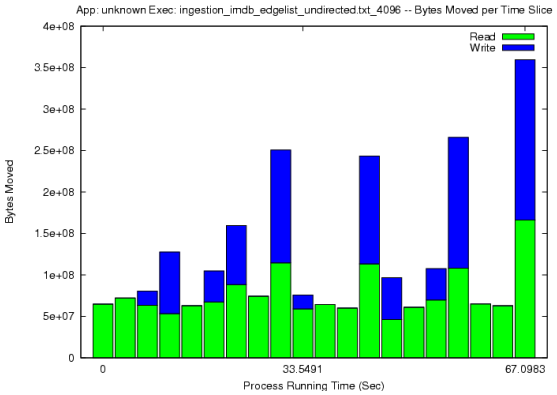
The results show that for some configurations, the benchmark is indeed I/O-intensive. This is particularly true for the search portion of the benchmark, where the code spends as much as 70% of its time for the I/O operations when accessing NFS-mounted data. Even when the data is stored in local storage devices for which there is no network communication overhead involved, the search code spends about 20% or more of its time to access externally stored data.

Figure 3.4, which presents the volume of the data read and written during run-time on local disk and Fusion-io, captures the I/O behavior of the benchmark. In this experiment, 4 KB blocks are used. The graph shows that the ingestion phase of the benchmark exhibits a regular access pattern. This is due to the systolic way in which the ingestion code operates; it repeats the process of reading, processing, and dumping the data. On the other hand, the search phase of the benchmark has a more irregular access pattern. This is because the search involves more files to access at the same time compared to the ingestion. This greater number of accesses increases the interference among concurrent file I/Os, and in turn, increases the I/O overhead. In addition, the search requires more reads than writes. Such behavior should favor the Fusion-io over the local disk, because the read from the Fusion-io is orders of magnitude faster than that from a typical hard disk. This is clearly shown in Table 3.1.

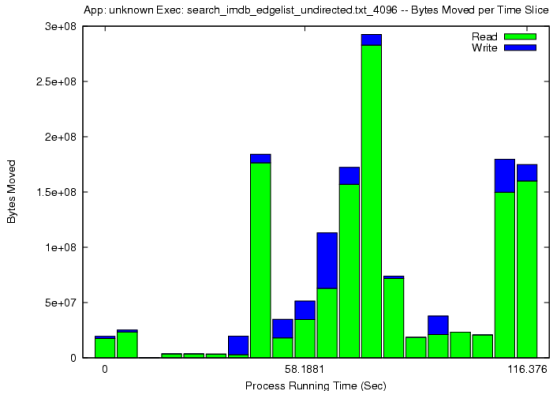
Figures 3.5 and 3.6 measure the impact of the block size on I/O performance. Two factors that affect the I/O performance of an application are the block size and the locality of the application. Graph applications usually have poor locality, but in the benchmark, the negative impact of the poor locality is reduced by sequentializing the accesses. When data is accessed from a network storage device, the block size affects the performance significantly due to the large overhead accessing the data. The effect of block size for network storage device was not shown in the figures. The figures show that the block size has little impact on the read and write performance except for the small block size (256B), which increases the number of I/O operations. Again, the search performance on Fusion-io is better than on local disk for aforementioned reason.

3.3 Conclusions

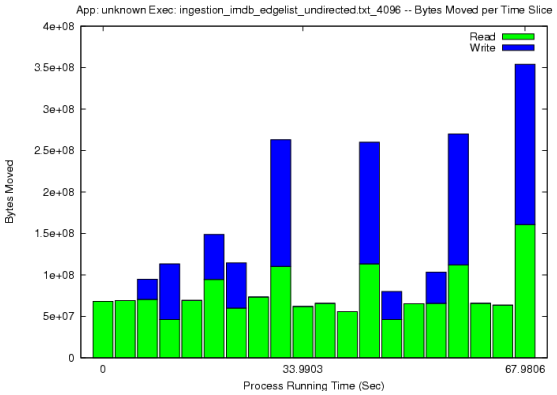
A graph application benchmark is developed in this work. The benchmark, based on a graph data management system called SGRACE, ingests streaming graphs and performs search in the form of level-set expansion. We have evaluated the performance of different storage devices using the benchmark and presented results.



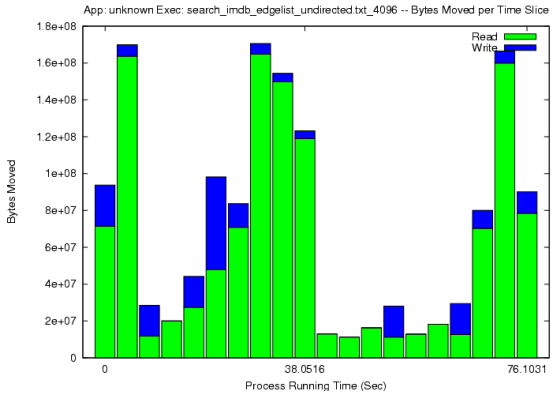
(a) Ingestion on local disk



(b) Search on local disk

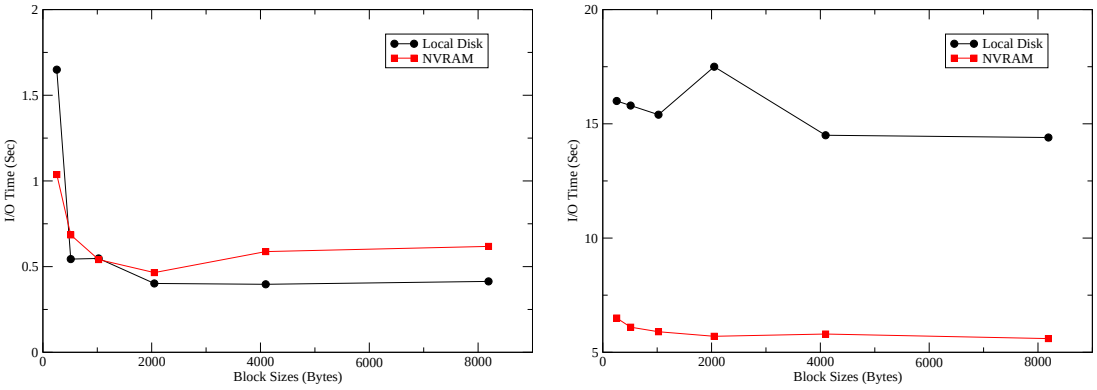


(c) Ingestion on Fusion-io



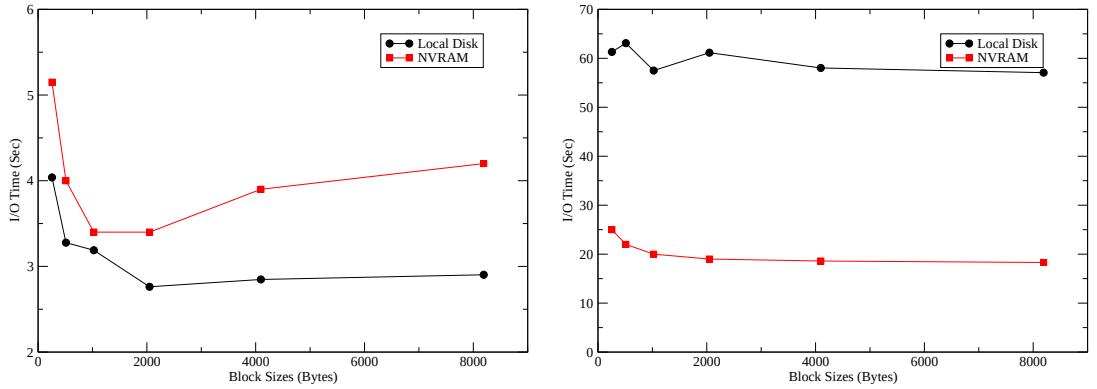
(d) Search on Fusion-io

Figure 3.4: Ingestion and search performance of the benchmark for the IMDB graph when data is accessed from local disk and Fusion-io.



(a) Ingestion on local disk (b) Search on local disk

Figure 3.5: I/O performance of benchmark on local disk for DBLP citation graph



(a) Ingestion on local disk (b) Search on local disk

Figure 3.6: I/O performance of benchmark on local disk for the Internet Movie Database (IMDB) graph

Chapter 4

The Livermore Entity Extraction Benchmark

4.1 Introduction

The Livermore Entity Extraction (LexTrac) pipeline addresses a data intensive problem of interest to national security: the automatic identification of references to entities, such as people, organizations, and locations, in free (a.k.a, unstructured) text. Entity extraction is an essential early step in populating the various Semantic Graphs that are increasingly being used by analysts for knowledge discovery and Homeland Defense.

There is, and will continue to be, a need to ingest and analyze millions of free text documents daily. This need far outstrips analysts' abilities to examine such documents manually. LLNL has been positioning itself to address these needs, and has committed itself to ongoing research and development in this area. However, as far as we are aware, little to no effort has been devoted to understanding how entity extractors and related applications, such as relationship extractors, interact with file systems. This is true not only at LLNL, but in the broader academic and open-source arenas. Painting with broad brush-strokes, the larger community is mostly concerned with developing and testing algorithms for automatic content extraction that work with a high enough degree of accuracy to be useful. In contrast, our interest here is to understand the I/O patterns of typical Natural Language Processing (NLP) applications. This knowledge will be essential as the NLP field matures, and enters the realm of storage-intensive supercomputing.

LexTrac is suitable as an NLP I/O benchmark for several reasons. First, as discussed below, it is an actual application that has been deployed in several projects, both inside LLNL and externally. Second, it was designed and implemented to fill an immediate need, hence, was not highly optimized for I/O performance. In this sense it is typical of the vast majority of extant NLP applications. Finally, LexTrac is not a monolithic code. Rather, it is a set of modules tied together by the file system. While some modules were developed at LLNL, others derive from the Open Source world. Each module can be examined independently, and each has distinct I/O characteristics. Therefore, leveraging LexTrac as a benchmark provides us with a suite of benchmarks, rather than a single kernel or application.

4.2 Background

Prior to presenting results for LexTrac I/O usage patterns, we discuss LexTrac's design and functionality, and present background material. Lextrac consists of a number of sequentially executing modules that (1) pre-process XML-encoded documents; (2) perform entity extraction on free text; and (3) compute statistics concerning those documents and the extracted entities.

Entity extraction is defined in Wikipedia as:

> ... a subtask of information extraction that seeks to locate and classify atomic elements in text

into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc. [1]

The process of entity extraction is referenced in the literature by a variety of names, e.g, *Named Entity Recognition (NER)*, *Entity Detection and Recognition (EDR)*, *entity tagging*, *Semantic Tagging*, and *Entity Identification*. We use the term *Entity Extraction* throughout this report.

Most researchers categorize entity mentions into three types[1]

1. **Named Entities** are proper names, such as *George Miller* or *Lawrence Livermore National Laboratory*;
2. **Nominals** are descriptive words or phrases, such as *the director* or *the laboratory*;
3. **Pronouns** such as *he*, *they*, *it*.

The canonical example of *free text* is newswire articles, and the experiments reported herein took as inputs XML-encoded Reuters News stories from the Reuters Corpora [2].

4.3 LexTrac Historical Design and Efficiency

The modules that comprise LexTrac derive from a variety of sources. Two of the modules, the *Stanford NER* [3] and the *OpenNLP NameFinder* [4] are Open Source codes. The *Keyword Extractor* was originally developed by David Buttler at LLNL. It, along with the UIMA wrappers we developed for the Open Source codes, have been deployed internally (at LLNL) and externally in a variety of configurations. Additional modules, such as the *Tabulator* and *Proximator*, were developed as the need arose and integrated into the LexTrac pipeline.

4.4 Module Descriptions

In this section we present specifics on the modules that comprise the Lextrac Pipeline. All modules are coded in Java. Each module is actually run as a separate UIMA [5] [2] pipeline executable.

1. The **XML Parser** is a simple hand-rolled parser that assumes the input XML documents are both well-formed and valid.
2. The **Keyword Extractor** is a module developed by David Buttler at LLNL for efficient extraction of words and phrases from a user-defined lexicon.
3. The **Stanford NER [3]** is a statistically based, Conditional Random Field (CRF) extractor; primary author is Jenny Finkel, Stanford PhD student.
4. The **OpenNLP NamFinder [4]** is a statistically based maximum-entropy based extractor; primary author is Tom Morton.
5. The **Unifier** is a "voting" module that, based on a comparison of the outputs of the three Entity Extractors, outputs a confidence measure for each extraction.
6. The **Proximator** computes proximity information for entities within a document, i.e, how many words separate each pair of extracted entities.

[1] An *entity mention* is a word or phrase in a text document that refers to a real world entity.

[2] The Unstructured Information Management Architecture (UIMA) originally developed by IBM, is a Java-based infrastructure that supports the implementation, description, composition, deployment, and reuse of UIMA components and applications. It contains a pipeline that consists of three components: a file reader; an analysis engine that wraps the essential modules functionality; and a file writer. UIMA is used in several projects at LLNL.

7. The **Tabulator** computes statistics such as frequency of punctuation marks, proportion of whitespace in a document, percentage of upper-case characters, etc.

8. The **Decider** uses the output from the Tabulator to make a yes/no decision as to whether a document most likely consists of sentence/paragraph free text, or tabular-like material such as stock market prices or sports scores.

LexTrac was envisioned as a modular pipeline, such that additional entity extractors and related modules could be easily inserted or deleted. Additional modules that have been used in LexTrac, but are not part of this benchmark, include the LingPipe [6] and Oak Ridge [12] entity extractors, and a TrueCasing module (designed to proper casing to all upper-case message traffic).

4.5 General I/O Considerations

The XML-parser reads a file from the input data set into memory; identifies the sections of interest to LexTrac; deletes any XML and HTML markup within those sections, and writes the result to an output file. In addition to extracting the free-text section (which is what the Entity Extractors will take as input), additional sections such as headline, byline, dateline, etc. are saved.

All subsequent modules read the output file into memory; request one or more sections from the file; perform their computations; append the result to a new section at the end of the file; and write the entire output file back to disk. Hence, as the pipeline progresses, the output files increase in size.

4.6 Data set Description

The input data set for these experiments consisted of 3183 XML-encoded Reuters News stories from the Reuters Corpora. [2] File sizes ranged from 1058 bytes to 33158 bytes. Figure 4.1 shows the distribution of file sizes. Except for a few outliers, most files were under 10K bytes. Collectively, the input data set contained 9.84M bytes. On conclusion of all processing, the output data set contained 49.4M bytes.

4.7 Performance Evaluation

We ran the LexTrac benchmark to profile its I/O behavior on a Linux machine equipped with the Fusion-io board, and captured I/O performance via the *iotrace* tool (described in Chapter 2.1). Runs were performed on three file systems: (1) an NFS (network) disk; (2) a local disk (/tmp); (3) the FLASH-based Fusion-io board. Tables 4.1 and 4.2 contain summary statistics for execution timing and I/O operations.

Total execution time for the NFS disk was approximately 464 seconds. Total execution times for the Fusion-io and local disks were 303 and 304 seconds respectively.

Per Table 4.2, the benchmark reads a total of approximately 500M bytes, writes just over 200M bytes, for a read/write ratio of 2.42. Figure 4.2 contains a plot of cumulative bytes read/written with respect to time. In this figure, as in others, the vertical line segments are artifacts we introduced during plotting in order to indicate the boundaries of the eight modules. The rapid increase in bytes moved during the last four modules has ramifications for the pipeline's future development. Until undertaking this study we did not realize the degree to which the policy of each module reading the entire output file, appending, and writing back to disk, was affecting I/O. In the future, when and if LexTrac is funded for additional development, the result of these experiments will certainly be taken into account.

Figure 4.7 accounts for all execution timing that was not involved in I/O activity. Since the plots for the local and Fusion-io disks are indistinguishable, we present only the plot for the Fusion-io disk. There is a noticeable "flattening" of the curve on both the left and right-hand portions of the plot for the local disk. This is indicative of portions of the benchmark where I/O activity begins to consume greater proportions of execution time.

metric	Execution Timing (sec)		
	NFS	local fs	Fusion-io
total	463.78	303.97	302.99
non I/O	307.67	299.87	300.34
I/O: read	1.25	0.93	0.69
I/O: write	0.39	1.59	0.67
I/O: open, close	154.48	1.59	1.27
	Timing Ratio (percent)		
I/O to non-I/O	50.7	1.4	0.9
read to write	321.4	58.4	103.0

Table 4.1: Summary timing statistics for the three filesystems tested. As expected, non-I/O timing is similar for all three. By far the most significant difference in I/O performance was not, as expected, in reads or writes, but in opening and closing files.

Statistic	Bytes
total read	494.7M
total write	204.4M
read/write ratio	2.42
avg read size	3323
avg write size	5053

Table 4.2: Summary I/O statistics. The numbers in this table are independent of any particular file system.

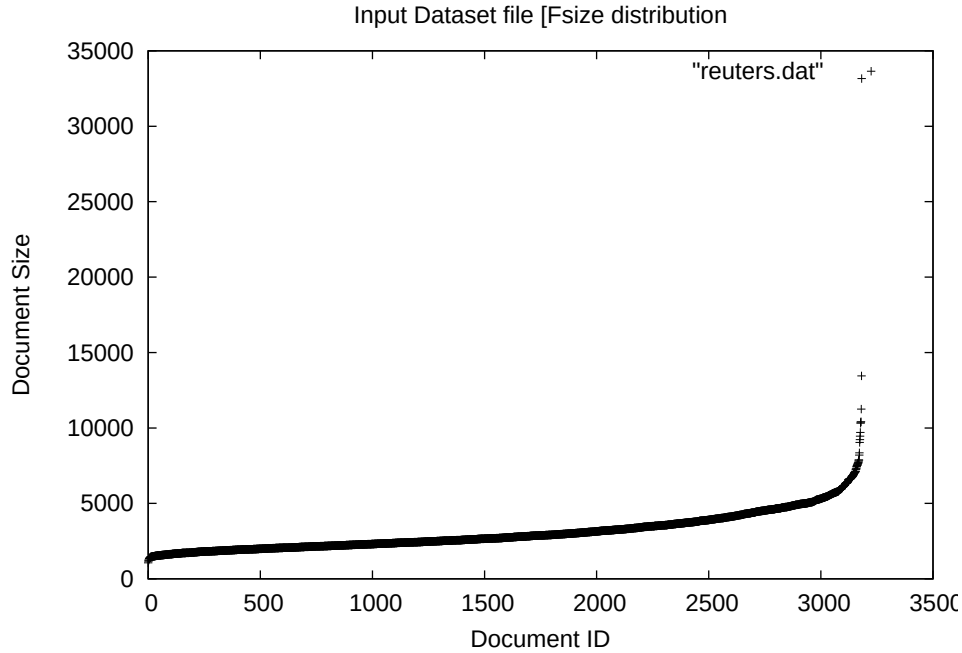


Figure 4.1: Input Data set size distribution. There is one data point per file.

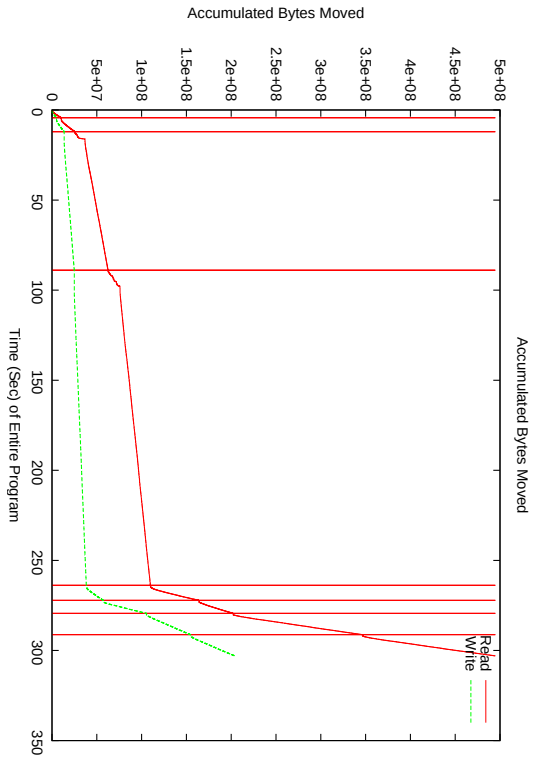


Figure 4.2: Cumulative bytes read/written.

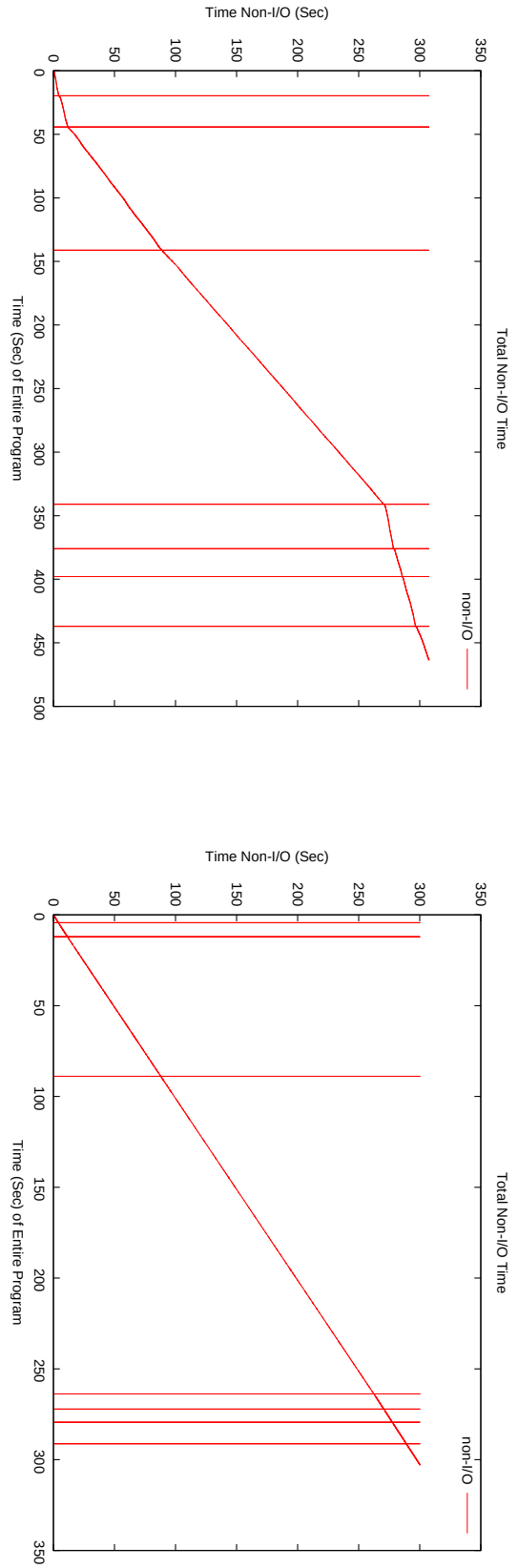


Figure 4.3: Non-I/O timing for NFS (top), and the Fusion-io (bottom).

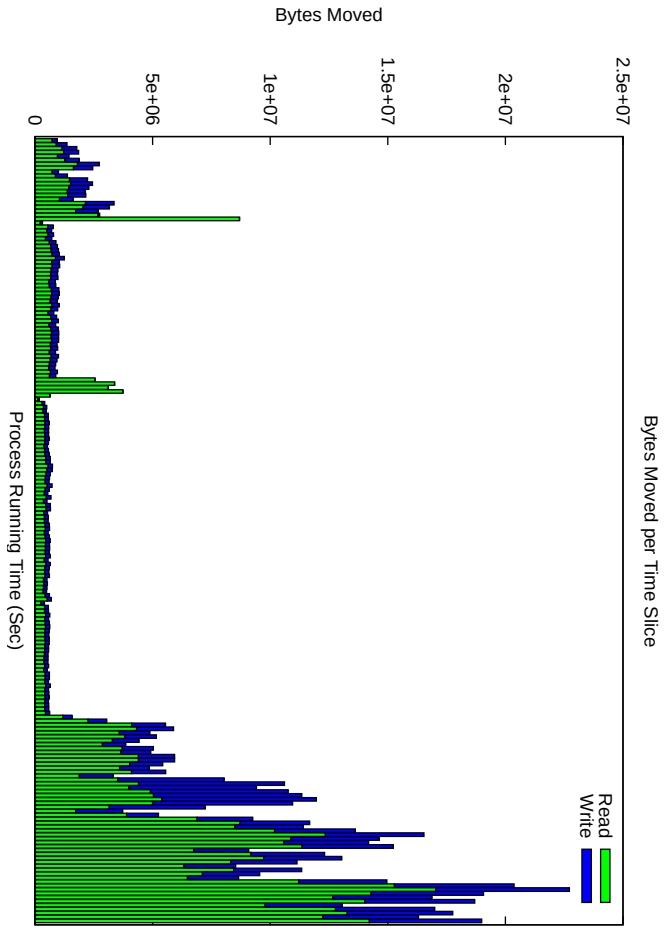


Figure 4.4: Bytes moved per time-slice, NFS. The x-axis represents 464 seconds. The figure contains 200 time slices, each of which is 2.32 seconds in width. Note that the y-axis indicates bytes moved per time slice; multiply the numbers on the y-axis by 0.66 to convert to bytes moved per second.

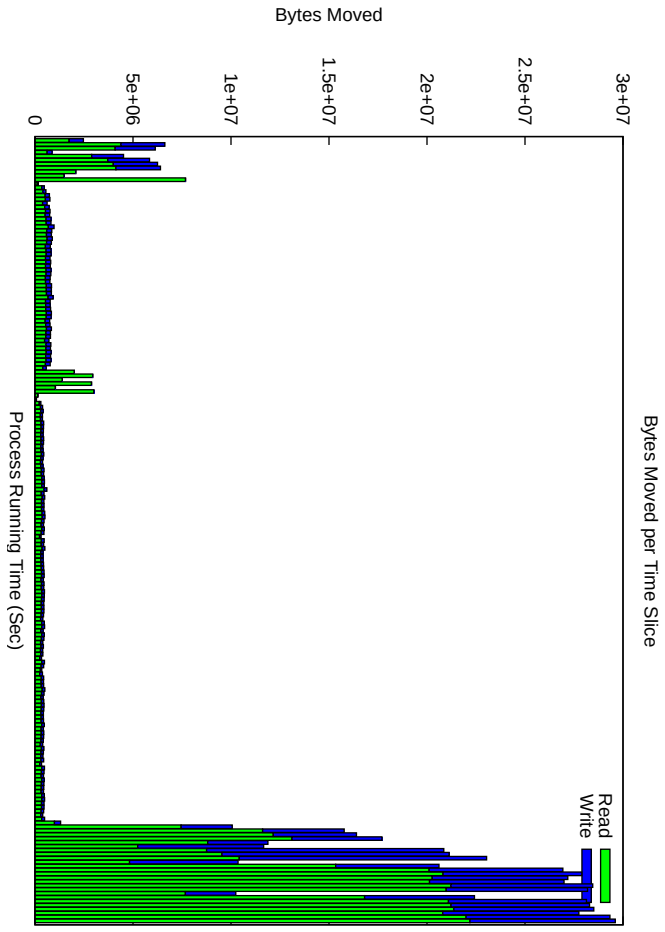


Figure 4.5: Bytes moved per time-slice, Fusion-io. The x-axis represents 304 seconds. The figure contains 200 time slices, each of which is 1.52 seconds in width. Note that the y-axis indicates bytes moved per time slice; multiply the numbers on the y-axis by 0.43 to convert to bytes moved per second.

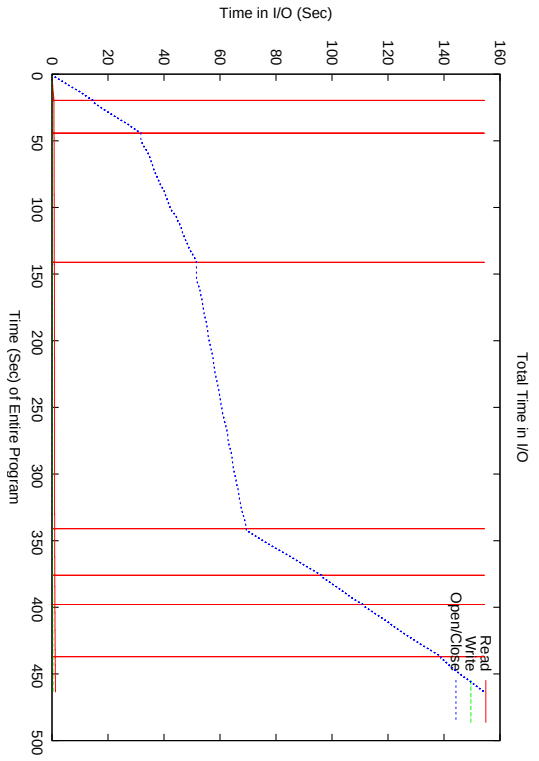


Figure 4.6: I/O time, NFS. I/O time is dominated by calls to open and close.

Figure 4.4 and 4.5 contain histograms showing the number of bytes moved per time-slice. Since the graphs for the Fusion-io and local disks are quite similar, we only present the graphs for the NFS and Fusion-io disks. These figures contain 200 time-slice bins. Since total execution time for the NFS run (x-axis) was 462 seconds, each bin represents 2.32 seconds of execution time. Since the y-axis indicates bytes moved per time-slice, the numbers on the y-axis should be multiplied by 0.43 to give an indication of bytes per second. Similarly, total execution time for the Fusion-io run was 304 seconds, each bin represents 1.52 seconds of execution time, so the multiplier to adjust to bytes per second is 0.66.

After adjusting to bytes per second, it appears that the Fusion-io board outperforms the NFS disk by roughly a factor of two, for the I/O intensive portions of the benchmark. Note that the Fusion-io and local disk times were about the same, indicating the extent of performance impact observed using the NFS-mounted data sets.

The differing I/O usage patterns of the eight benchmark modules are clearly evident in the graphs. Starting from the left-hand-side, we see two bursts of I/O activity, which correspond to the XML-parser and Keyword Extractor. This is followed by a spike of read-only intensity, which is accounted for by the Stanford Extractor's initialization phase, in which a the 11M statistical model is read from file. This is followed by the extraction process; the lower levels of I/O activity are indicative of the primarily CPU-intensive categorization of this phase. Another spike of read-only intensity is accounted for by the OpenNLP's initialization phase, during which it reads in several statistical models. As for the Stanford Extractor, this initialization phase is followed by lower levels of I/O activity. Finally, near the right-hand-side of the graphs, there are four clusters of mixed read/write activity that correspond to the Unifier, Proximator, Tabulator, and Decider modules. These modules demonstrably have much greater read/write demands than the parser and extractors.

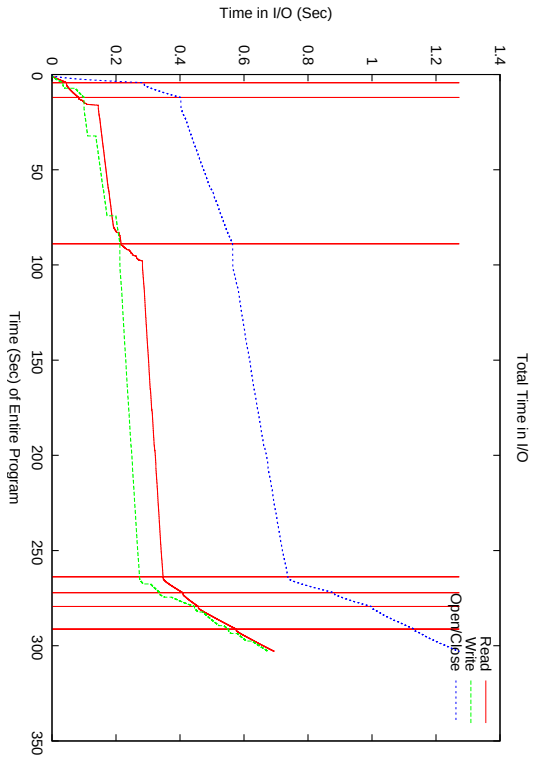


Figure 4.7: I/O time, Fusion-io. Compared to the previous figure, timing for calls to open and close is vastly reduced.

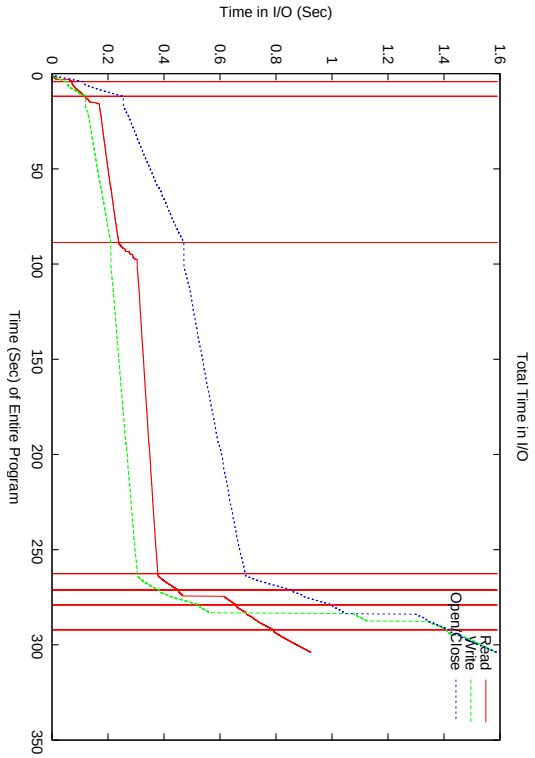


Figure 4.8: I/O time, local. Timing for the local disk is relatively similar to that for the Fusion-io drive, however, here there is a significant jump in time spent in write, and open/close calls in the last four modules. These four modules are by far the most I/O intensive of the benchmark. The jump in write/open/close calls is believed

Chapter 5

A GPU-Accelerated Image Resampling Benchmark

5.1 Application Domain

The Large Synoptic Survey Telescope (LSST) will generate 15 Gbytes of data every 15 seconds. This data must be analyzed in real-time to meet the science goals of the project. The image processing pipeline imposes significant computational and disk IO requirements on the analysis system. Each image is 9 GBytes.

The first significant step in the pipeline is called source detection. This is used to refine the exact position on the sky of the image. A source catalog of the sky is read from disk and multiple bright stars are used to get an accurate pointing. This accurate pointing is referred to as the world coordinate system.

The image is now rescanned using a threshold value to identify each group of contiguous pixels that exceed some threshold. Astronomical objects are not points and most objects overlap some other object. To separate these objects into distinct objects, a further sub-thresholding calculation is required. Once completed, a source catalog is read from disk and each detected object is modified to meet a predetermined brightness. Atmospheric effects always distort each object from its ideal appearance and a further calculation is performed on each object to make this correction. Finally, a 36 Gbyte template of the sky is read from disk and the modified image is subtracted from the corrected source image to determine what objects have changed. Each changed object is classified into types such as a known variable star, a supernova or nova, a known moving object like an asteroid, or a newly discovered asteroid. The classification is done by comparing these objects to known objects and their characteristics in a database. The computational load requires a 27 Teraflop system capable of reading about 50 GBytes of data every 15 seconds.

5.2 Benchmark Implementation

The benchmark factors out a computationally-expensive and fundamental piece of the SWarp [8] functionality for implementation: greyscale image resampling. The input data is a greyscale raster-image (row-major order) with 8 or 16 bits per pixel. The output is a greyscale image with 8, 16, or 32 bits per pixel. The typical SWarp execution for LSST would include resampling of a 16-bit input into a 32-bit, floating-point output.

5.2.1 Computational Kernels

For each output pixel, a filter kernel is applied that takes a weighted combination of the input pixels. The SWarp code employs a Lanczos filter to combine either 16 (4x4), 36 (6x6), or 64 (8x8) input pixels to generate each output pixel.

kernel	TEX1	SIN1	RCP1	MUL2	ADD2	MAD1	MUL1	FLR2	VTot	STot
Nearest	1	0	0	0	1	0	0	1	2	3
Lanczos2	16	16	16	25	21	15	16	1	126	173
Lanczos3	36	24	24	49	43	35	24	1	236	328
Lanczos4	64	32	32	81	73	63	32	1	378	533

Table 5.1: Fragment processor operation counts (ordered roughly from most expensive to least expensive) for each kernel.

The Lanczos filter kernel is convolved with the input pixels to generate the output pixels. Each input pixel has a location (x, y) relative to the projection of an output pixel into the input image. The weight used for that input pixels contribution to the output pixel is:

$$L_k(x,y) = \frac{k\sin(\pi x)\sin(\frac{\pi}{k}x)}{\pi^2 x^2} * \frac{k\sin(\pi y)\sin(\frac{\pi}{k}y)}{\pi^2 y^2}, |x| < k, |y| < k \tag{5.1}$$

k may be 2, 3, or 4 (the so-called Lanczos2, Lanczos3, and Lanczos4 kernels), using 16, 36, or 64 input pixels per output pixel, respectively.

We have written four resampling codes, one for each of the above Lanczos convolution kernels, as well as a baseline nearest-neighbor resampling using NVIDIA's Cg programming language. The Cg compiler compiles the code down to ARB Fragment Program code, which is an OpenGL-supported assembly code for the fragment programs on current GPUs, and is vendor-neutral. The operation counts in ARB Fragment Program assembly for our implementation of each of these resampling codes is shown in Table 5.1. The operations listed are:

- **TEX1**: Read one input pixel value from texture memory.
- **SIN1**: One sine computation
- **RCP1**: One reciprocal computation
- **MUL2**: Two-way-vectorized multiplication
- **ADD2**: Two-way vectorized addition
- **MAD1**: One multiply-and-add computation
- **MUL1**: One multiplication
- **FLR2**: Two-way vectorized floor computation
- **VTot**: Number of operations for a vector processor
- **STot**: Number of operations for a scalar processor

Because we are benchmarking the recent NVIDIA 8800 GTX, which is a scalar GPU, we have not optimized the computation for the more traditional 4-way vectorized GPU, which typically can do 4-way operations at the same cost as a single operation. The philosophy behind the NVIDIA 8XXX series is to provide more scalar cores (128) rather than fewer vector cores, so it is easier to achieve full utilization.

5.2.2 I/O Methods

In the SWarp implementation, which employs memory-mapped I/O for large files, output pixels are processed in raster order, matching the order of the data in the disk file, so output is purely streaming, whereas input requires more random access.

However, to take best advantage of the significant processing power, memory bandwidth, and data caching on the GPU, it is more natural to produce the output data in 2D *tiles*. Each output pixel in the tile is generated in a separate thread of execution on the GPU.

Data flow in this benchmark is from disk to main memory, main memory to GPU memory, GPU memory back to main memory, and main memory back to disk. In each case, the data to be moved from source to target location is specified as a 2D rectangle. If the width of this rectangle is the same as the width of the source and target, the data copy can occur as a single, contiguous stream. However, if the widths differ, then the data copy must occur in a *strided* fashion, contiguous only at the level of individual rows. Strided copies incur additional overheads, the largest of which are disk seeks if the striding occurs on the disk file end of a copy between main memory and disk. Striding can also incur more minor overhead when it occurs between main memory and GPU memory. Note also that support for strided copies is generally built into GPU drivers, whereas it is not built into the operating system's file I/O interface and must be built using some combination of reads, seeks, and buffering.

The computation proceeds one output tile at a time, in row-major order. For each output tile, we determine the input data required and load that rectangle of data from main memory to the GPU over the PCI-Express bus. We then perform the computation, and read the data back from GPU to main memory over PCI-Express.

We have implemented four I/O methods for moving data between the disk and main memory.

1. **Basic**: This method assumes the input and output data are small enough to fit in-core. Thus, the entire input data are read with a single read and the entire output data written after the tiled computation is performed. Note that the GPU typically has less memory than the host system, and there is also a hard limit on the maximum 2D image size it can operate on. Strided copies are used to move data between main memory and the GPU memory.

2. **MMap**: This method uses the operating system's memory mapped I/O. A blank output file of the appropriate size is created on disk, then both the input and output files are memory mapped. The operating system pages data in on-demand and flushes pages as the buffer cache becomes full. This method appears much like the Basic method, because the entire input and output files are virtually present in main memory. Thus we still use strided copies between main memory and GPU memory.

3. **Block**: This method manually loads the appropriate input rectangle for a given output tile using a sequence of seeks and reads, one for each row. Similarly, it writes each output tile as a sequence of seeks and writes. Thus, strided copies are used to and from disk. The data is contiguous in main memory, so strided copies are not needed between main memory and the GPU.

4. **Strip**: This method loads and stores the disk files using *strips*, where a strip is a rectangle with the same height as an input or output tile, and the width is the same as the width of the entire input or output image. It is effectively a buffered version of the Block method, where the buffer is all the blocks in a single row of output tiles. Because the entire row is buffered, the file data is now contiguous may be read or written with a single file operation. GPU tile data transfers are once again strided.

5.3 Performance Evaluation

We have used our benchmark to investigate the potential for accelerating image resampling for applications such as the LSST project. The test system has the following specifications:

- **Operating System**: Fedora Core 6 Linux x86_64, kernel 2.6.22.2-42.fc6
- **CPU**: Dual Intel Xeon 3.0 GHz
- **Memory**: 4 GB
- **GPU**: NVIDIA GeForce 8800 GTX

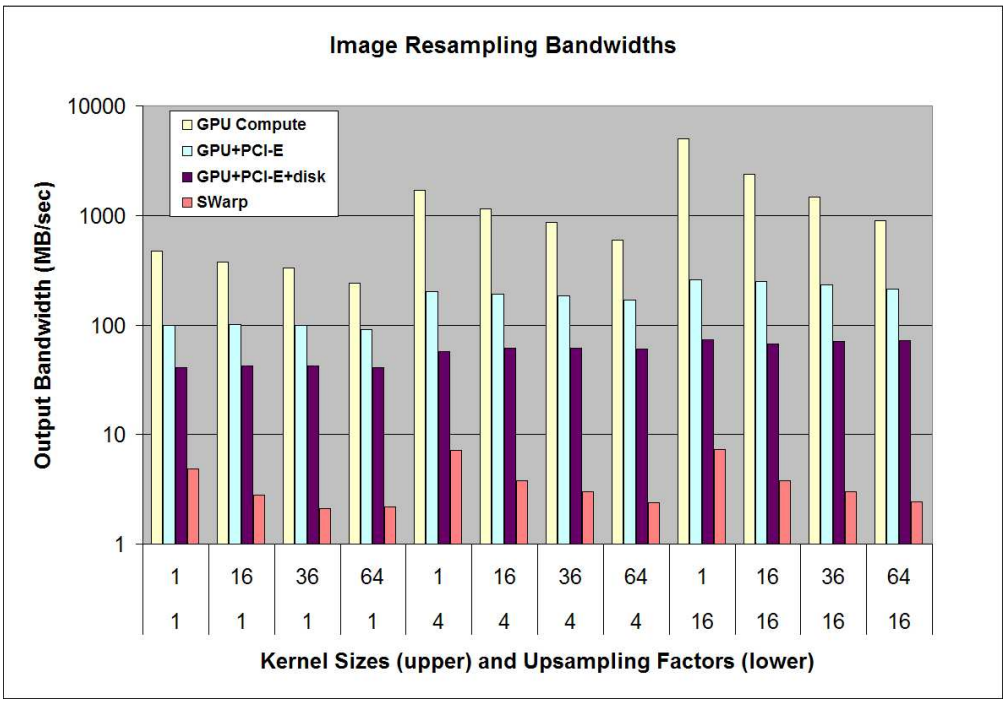


Figure 5.1: Rate of generated output pixels in logarithmic scale as a function of upsampling scale factor and kernel size

- **GPU Driver**: NVIDIA 100.14.11
- **GPU Memory**: 768 MB
- **GPU Bus**: PCI-E 16X
- **Disk**: 2-disk RAID0, local SATA, ext3 filesystem

Our investigation comprised a number of executions of the benchmark and the original SWarp application. The most informative metric reported for this application is the rate of data output in MB/sec. The quantitative results are shown in Table 5.2 and plotted in Figures 5.1 and 5.2.

For each execution, we choose a scale factor, which is the ratio of output pixels to input pixels, and a kernel size, which is the number of input pixels combined to generate each output pixel.

The GPU column indicates the pure computational rate of our GPU-based kernel implementation. This is measured by running the Basic (in-core) method for disk I/O and disabling data transfers to and from the GPU and measuring the total compute time by issuing commands to the GPU and then flushing the pipeline to force it to complete execution. It is not surprising that this rate decreases with increasing kernel size (number of instructions as a function of kernel size are shown in Table 5.1). What may be less obvious is the dramatic increase in compute performance as the scaling factor is increased. The is attributable to the highly effective 2D data (texture) cache on the GPU. As the scale factor is increased, the same neighborhood of input pixels are used over and over again to generate a neighborhood of output pixels.

The GPU+PCI-E column reports the performance when we now take into account data transfers to and from the GPU over the PCI-Express bus. The time to read data back from the GPU to the host generally dominates for two reasons: (1) there is more data to read back from the GPU than to send, due to the upsampling factor and the doubling in bits per pixel (from 16-bits to 32-bits) and (2) raw download rates are typically slower than upload rates in general. In truth, we are not achieving the fastest possible

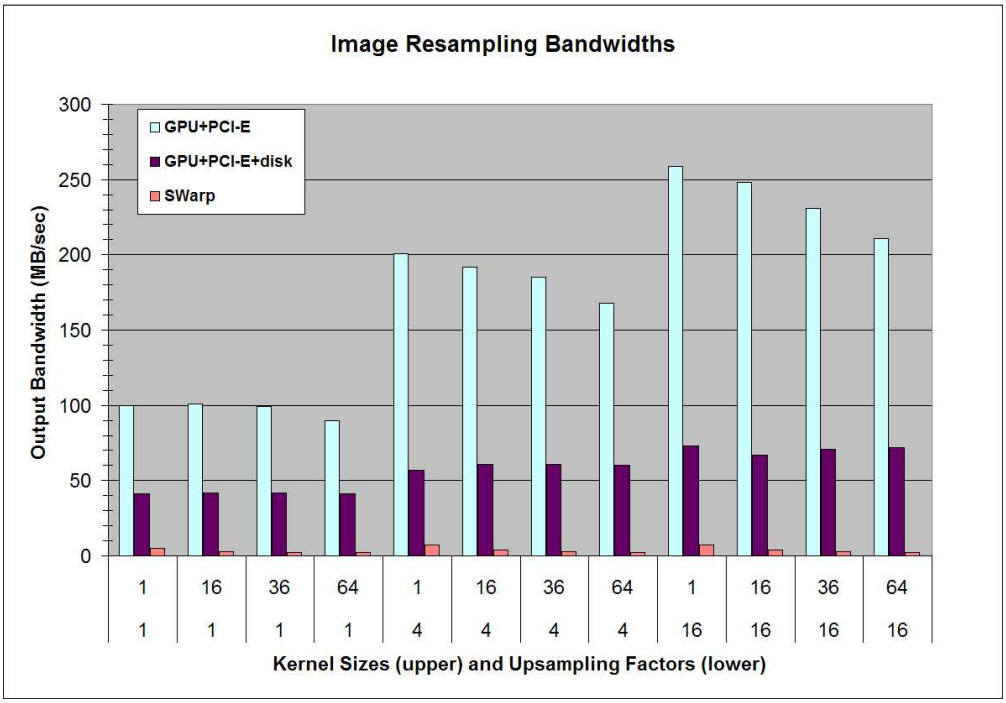


Figure 5.2: Rate of generated output pixels as a function of upsampling scale factor and kernel size

Scale	Kernel	GPU	GPU+PCI-E	GPU+PCI-E+disk	SWarp
1	1	475	100	41	4.8
1	16	372	101	42	2.8
1	36	329	99	42	2.1
1	64	239	90	41	2.2
4	1	1706	201	57	7.2
4	16	1143	192	61	3.8
4	36	869	185	61	3.0
4	64	598	168	60	2.4
16	1	4994	259	73	7.3
16	16	2356	248	67	3.8
16	36	1475	231	71	3.0
16	64	896	211	72	2.4

Table 5.2: Output bandwidth, reported in MB/sec, as a function of the upsampling scale factor and kernel size.

PCI-E transfer rates in our current benchmark implementation. Due to the intricacies of GPU and driver optimizations, experiments have shown we can improve PCI-E by roughly a factor of 2X by downloading data as a 4-channel (RGBA) image rather than a 1-channel (luminance) image. This optimization could be implemented, for example, by computing 4 horizontally-adjacent output pixels at each thread execution. However, even if we incorporated such speedup into our benchmark, it could not yet be leveraged due to the disk I/O bottleneck discussed in the next paragraph.

The GPU+PCI-E+disk column reports the true overall performance of the benchmark on out-of-core data, using the Strip method of disk I/O (in practice, we have found the Strip method performs comparably or slightly better than MMap and dramatically better than Block). The maximum output performance achieved is 73 MB/sec. For the lowest scale factor of 1, this performance is significantly reduced (to around 40 MB/sec), presumably because we require significant input data bandwidth as well as output data bandwidth, and the input and output files are sharing the same 2-disk RAID (experimentally disabling reading of input data from disk in this case increased performance to around 70 MB/sec).

To measure the performance of the real SWarp application, we eliminated as many computations as possible that are extraneous to the benchmark computation. In particular, the resampling is configured to perform in SWarp's PIXEL coordinate system, thus avoiding any of the dozens of more complex astrometric coordinate transformations. It is possible that SWarp is still performing some additional computations that could not be simply disabled, so direct comparison here should be taken with a grain of salt (however, we did find that SWarp executed somewhat faster than running the benchmark using the Mesa OpenGL driver for software-only execution, so the timing of SWarp is not implausible for a CPU-only implementation). The range of speedups of the GPU implementation compared to SWarp ranged from 9X to 30X. The need for oversampling and high-quality kernels to produce excellent results argues that the test with the larger speedups are the more useful ones in practice. SWarp can also be executed in parallel on multi-processor and multi-core machines, and exhibited a nearly 2X speedup over the listed results when run on the two processors of the test machine.

Notice that the original SWarp implementation is purely CPU-bound, where the GPU benchmark is more thoroughly I/O-bound. Thus there is significant room for further speedup if faster I/O were available (such as the 600 MB/sec write speed recently reported in the news for the Fusion IO PCI-Express NAND flash memory board). Such faster I/O can bring the speedup closer to the 87X of our current GPU+PCI-E speedup over SWarp (or the projected 160X speedup of a GPU+PCI-E with the aforementioned 4-channel readback optimization).

5.4 Conclusions

The image resampling benchmark demonstrates that the use of commodity parallel architecture such as the GPU can provide dramatic speedup over a single-CPU implementation, up to the point where disk I/O becomes the primary bottleneck. For the current test system, the I/O limited the speedup to 30X.

In general, it is clear that providing faster out-of-core I/O, especially in terms of write speed for this application, will enable even more dramatic speedups overall.

For the specific case of LSST software pipeline, the best solution may be to engineer a streaming, out-of-core data model in which such large and upsampled intermediate data never needs to be written. Thus performing the subsequent computations following resampling, performing data analysis and database queries, which ultimately reduce the size of output data, would be preferable from a bandwidth point of view, though it may complicate the software design. It may well be possible to apply commodity parallel architectures to many stages of the LSST pipeline, making the real-time goals of the project achievable.

Bibliography

[1] http://en.wikipedia.org/wiki/Named_entity_recognition.

[2] http://trec.nist.gov/data/reuters/reuters.html.

[3] http://nlp.stanford.edu/ner/index.shtml.

[4] http://opennlp.sourceforge.net/index.html.

[5] http://www.alphaworks.ibm.com/tech/uima.

[6] http://www.alias-i.com/lingpipe/.

[7] Barabasi, A.-L., and Albert, R. Emergence of scaling in random networks. *Science 286* (1999), 509.

[8] Bertin, E. SWarp v2.16.4 user's guide. Tech. rep., Institut d'Astrophysique & Observatoire de Paris, July 2007.

[9] Business information system KOMPASS. http://www.kompass.com/ip.

[10] Computer Science Bibliograph. http://dblp.uni-trier.de/xml/ .

[11] Karavanic, K. L., May, J., Mohror, K., Miller, B., Huck, K., Knapp, R., and Pugh, B. Integrating database technology with comparison-based parallel performance diagnosis: The perftrack performance experiment management tool. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2005), IEEE Computer Society, p. 39.

[12] Symons, C. T., Samatova, N. F., Krishnamurthy, R., Park, B. H., Umar, T., Buttler, D., Critchlow, T., and Hysom, D. Multi-criterion active learning in conditional random fields. In *ICTAI '06: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 323–331.

[13] The Internet Movie Database (IMDb) graph. http://www.iruimte.nl/graph/imdb.txt .

[14] Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., and Ümit Çatalyürek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of Supercomputing'05* (Nov. 2005).